1. Exercices Java

Table des matières

Exercice 1 : Que suis-je ?

Pour les exemples suivants, déterminer s'il s'agit d'une classe, d'un objet, d'un attribut ou d'une méthode. S'il s'agit d'un objet, vous mettez un X dans la case « Objet », et vous mettez le nom de la classe dans la case « Classe ». S'il s'agit d'une classe, vous mettez le nom de la classe dans la case « Classe » et vous laissez la case « Objet » vide.

	Nom de la classe	**Objet**	**Attribut**	**Méthode**
paires de lunettes				
hurler				
le crayon de Paul				
couleurs				
la pomme que ma mère m'a donnée ce matin				
ordinateurs				
détruire				
planètes				
singes				
la couleur de mon pullover				
le cours que je suis en train de suivre				
microprocesseurs				
Bill Gates				
la taille d'un client				
cet exercice				

Pour avancés :

Est-ce qu'il y a des exemples où l'on pourrait cocher plusieurs cases ? Si oui, lesquelles ? Comment procéderais-tu pour décider dans ces cas ?

Exercice 2 : Figures géométriques - modélisation

Nous voulons développer un logiciel pour dessiner des formes géométriques. Quelles classes peut-on choisir ?

Quels sont les attributs et méthodes de ces classes qui nous intéressent dans ce contexte ?

La liste des classes et de leurs attributs et méthodes qu'on vient d'établir n'est pas unique. On aurait pu en choisir d'autres. En général on fait son choix sur base du problème posé, c.-à-d. on laisse de côté toutes les classes et caractéristiques qu'on ne juge pas nécessaires pour la résolution du problème.

Exercice 3 : Figures géométriques en pratique

1. Ouvrez le projet **`Shapes`** avec Unimozer. Chacun des rectangles colorés représente une classe du projet : **`Circle`**, **`Square`**, **`Triangle`** et **`Canvas`**.
2. Cliquez du bouton droit sur la classe **`Circle`**.
3. Dans le menu déroulant qui s'affiche, choisissez *new Circle()*.
4. Acceptez le nom donné par défaut « **`circle0`** » en cliquant sur OK.
5. Vous venez de créer votre premier objet, que vous voyez en rouge en bas de l'écran.
6. Inspectez les attributs qui sont affichés avec l'objet et jouez avec les méthodes de l'objet qui deviennent visibles si vous cliquez du bouton droit sur l'objet.
7. Dans Unimozer, les attributs sont affichés dans les objets en bas de la fenêtre.
8. Changez la couleur du cercle, changez sa taille et déplacez-le.
9. Créez 2 autres cercles, ainsi que 2 triangles et 1 carré.
10. Expérimentez avec vos objets.

Remarque:

La classe **`Canvas`** permet de représenter les figures à l'écran. Vous n'avez pas besoin de vous occuper du code de la classe **`Canvas`**, vous pouvez même l'ignorer complètement.

Après avoir créé une ou plusieurs figures, rendez-les visibles en appelant la méthode '**makeVisible()**'. Les figures seront alors dessinées à l'écran, et vous pouvez suivre les modifications dans les dessins lorsque vous appelez les différentes méthodes des figures.

Exercice 4 : Branches

Définissez la classe **`SchoolSubject`** avec les attributs **`test1`**, **`test2`** et **`test3`** représentant les notes des 3 devoirs (nombres entiers) écrits dans cette branche.

Définissez la méthode **`setMarks`** (avec 3 paramètres) permettant de donner des valeurs aux trois notes.

Développez les méthodes **`getSum`**, **`getProduct`**, **`getAverage`** permettant de calculer

- la somme,
- le produit,
- la moyenne arithmétique

des trois notes.

Notions requises : classe, objet, méthode, paramètre, type

Exercice 5 : Figures géométriques

Définissez les classes servant à représenter les figures géométriques suivantes :

- carré,
- rectangle,
- cercle,
- cube,
- parallélépipède rectangle (angl : cuboid),
- sphère.

Les classes disposent de méthodes permettant de calculer le périmètre (DE : *Umfang*) et la surface; pour les figures à trois dimensions il faut pouvoir calculer la surface et le volume.

Pour chaque classe, choisissez judicieusement les attributs (nombres réels) qui sont nécessaires pour pouvoir effectuer les calculs demandés.

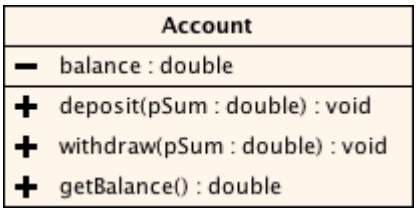
Chaque classe dispose d'un constructeur permettant d'initialiser les attributs.

Notions requises : classe, objet, méthode, paramètre, type, constructeur

Exercice 6 : Comptes bancaires

Développez la classe **`Account`** (FR : compte; DE : Bankkonto) qui possède les propriétés et fonctionnalités suivantes :

1. le compte a un solde (nombre réel)
2. au début le compte est vide (le solde est zéro)
3. on peut ajouter un certain montant à un compte
4. on peut retirer un certain montant d'un compte
5. on peut demander le solde actuel (accesseur)



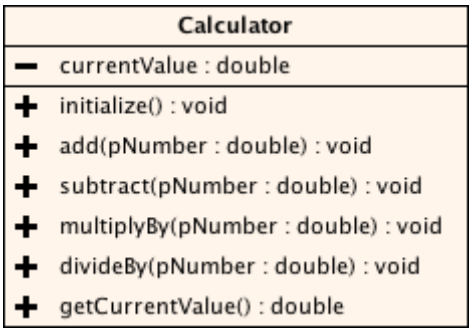
Remarque :

On suppose que les valeurs entrées pour **`pSum`** sont positives. **`balance`** peut prendre des valeurs négatives.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void

Exercice 7 : Calculator

1. Implémentez la classe **`Calculator`** représentée par le diagramme de classe suivant :



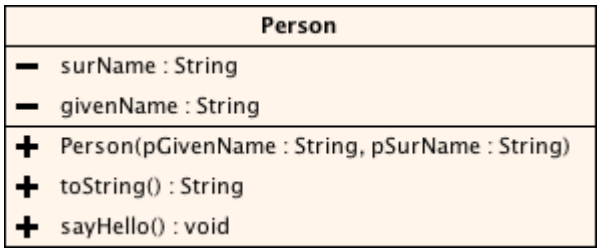
2. Ajouter maintenant l'attribut **`operations`** à votre classe. Cet attribut doit compter le nombre d'opérations de calcul effectuées par la calculatrice depuis la dernière initialisation.

   Afin de pouvoir vérifier si l'attribut **`operations`** fonctionne de manière correcte, ajouter aussi un accesseur **`getOperations`** à votre classe.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void

Exercice 8 : Person

A la naissance d'un enfant, ses parents lui donnent un nom. Écrivez donc la classe **`Person`** de la manière suivante :



- La méthode **`toString`** retourne le nom et le prénom de la personne dans une chaîne de caractères de la forme **`« <prenom> <nom> ».`**
- La méthode **`sayHello`** affiche dans la fenêtre message un texte de la forme **`«Hello, my name is <prenom> <nom>! ».`**

Notions requises :
classe, objet, méthode, paramètre, type, attribut, constructeur, println, type String

Exercice 9 : Body Statistics

1. Développez la classe **`BodyStatistics`** qui possède les attributs suivants d'une personne :
   - **`age`** l'âge (en années),
   - **`height`** la taille (en cm),
   - **`weight`** le poids (en kg).
2. Ajoutez un constructeur qui initialise les attributs. On suppose (sans vérification) que les données entrées (âge, taille, etc.) sont des données réalistes.
3. Selon la formule de Broca, le poids normal se calcule de la manière suivante :

   ```
   POIDS (kg) = TAILLE (cm) - 100
   ```

   Développez donc la méthode **`getNormalWeight()`** qui calcule le poids normal d'une personne suivant la formule de Broca !
4. Le poids idéal selon Broca est calculé de façon différente pour hommes et femmes :

   pour hommes : poids idéal = poids normal * 0,9

   pour femmes : poids idéal = poids normal * 0,85

   Développez deux méthodes **`getIdealWeightWoman()`** et **`getIdealWeightMan()`** qui calculent le poids idéal pour femmes et pour hommes suivant la formule de Broca !

Attention : Ces formules sont valables uniquement pour des adultes et elles donnent des résultats inadaptés pour des personnes très grandes, très petites, très musclées, etc.

5. L'index de corpulence, encore appelé «Indice de Masse Corporelle», ou en anglais "Body Mass Index" (BMI) donne de meilleurs résultats que les indices de Broca. Il se calcule de la manière suivante :

   `BMI = POIDS (kg) / TAILLE`^{2} **`(m`**^{2}**`)`**

   Programmez donc la méthode **`getBMI()`** qui détermine le BMI d'une personne !

Informations supplémentaires : Pour vous donner une idée du résultat, voici (à gauche) le tableau d'évaluation pour des personnes adultes. A droite un tableau montrant le poids normal au fil de l'âge. D'autres tableaux d'évaluation peuvent être retrouvés sur Internet.

Situation pondérale	BMI Femme	BMI Homme
Maigreur	<19.1	< 20.7
Poids normal/idéal	19.1 - 25.8	20.7 - 26.4
à la limite du surpoids	25.8 - 27.3	26.4 - 27.8
Surpoids	27.3-32.3	27.8 - 31.1
Obésité	> 32.3	> 31.1

Âge (ans)	Adaptation BMI normal
19–24	-2
25–34	-1
35–44	0
45–54	+1
55–64	+2
> 64	+3

Notions requises : constructeur, classe, objet, méthode, paramètre, type, attribut, void

Exercice 10 : Randomizer

Créez la classe **Randomizer** qui sert à générer des valeurs aléatoires entières dans des limites données.

La classe possède les attributs suivants :

- **min** et **max** les limites pour les valeurs aléatoires à générer
  (par défaut : min=0 et max=10)

La classe possède les méthodes suivantes :

- **setLimits(long pMin, long pMax)** ré-/initialise les attributs
- **Randomizer(long pMin, long pMax)** initialise les attributs
- **long getNext()** retourne comme résultat une valeur aléatoire entre **min** et **max** (limites incluses)
- On suppose que **pMin** est toujours inférieur ou égal à **pMax**, sinon la classe se trouve dans un état incorrect et la réaction de ses méthodes est imprévisible.
- **Pour avancés :** modifiez **setLimits** et **Randomizer** de façon à ce que le rôle de **pMin** et **pMax** soit échangé si **pMin** est supérieur à **pMax** (c.-à-d. l'attribut **min** n'obtiendra jamais une valeur supérieure à **max**).

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur, Math.random(), conversion de type

Exercice 11 : Devoir en classe

1. Créez la classe **Test**, qui mémorise une note (EN : mark) d'un devoir en classe et qui donne une évaluation de la note.
2. La classe dispose d'un attribut **mark** qui est initialisé dans le constructeur. On suppose (sans vérification) que les valeurs entrées pour **mark** sont des notes dans l'intervalle usuel [1...60].
3. Définissez l'accesseur **getMark** qui retourne la valeur de l'attribut **mark**.
4. Définissez la méthode **getEvaluation** :

   Si l'attribut **mark** a une valeur supérieure ou égale à 30 alors la méthode **getEvaluation** retourne le texte (type **String**) « test passed » sinon elle retourne le texte « test failed ».

   Créez un devoir et vérifiez le fonctionnement correct.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, conditions simples
Structures requises : if (simple)

Exercice 12 : Cistern

Il s'agit de développer une classe simulant une citerne d'eau potable. Développez la classe **`Cistern`** (DE : Wassertank) qui possède les propriétés et fonctionnalités suivantes :

1. une citerne possède un volume maximal **`maximumVolume`** de 1000 litres
2. une citerne a un volume actuel **`currentVolume`**
3. on peut ajouter de l'eau à la citerne (méthode **`add(double pVolume)`**)
4. on peut retirer de l'eau de la citerne (méthode **`drain(double pVolume)`**)
5. la citerne renseigne sur son volume actuel
6. la citerne renseigne sur son taux de remplissage (en pour-cent)
7. la citerne retourne un texte décrivant son état actuel (méthode **`toString()`**)

On suppose que **`pVolume`** est toujours positif. Les méthodes **`add`** et **`drain`** sont à programmer de façon à ce que **`currentVolume`** ne dépasse jamais **`maximumVolume`** et ne soit jamais inférieur à zéro. Comme en 'réalité', si on essaie p.ex. d'ajouter plus d'eau que la citerne peut contenir, elle sera pleine et le reste de l'eau sera perdue (ignorée).

Modifiez votre classe **`Cistern`** ! Cette nouvelle version ne possède plus un volume maximal fixe de 1000 litres. Ajoutez le constructeur **`Cistern(double pRadius, double pHeight)`** qui permet de créer des citernes à volume variable selon leurs dimensions extérieures. N'oubliez pas que le volume maximal ne peut plus être modifié après que la citerne est construite !

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur
Structures requises : if (simple)

Exercice 13 : Article

1. Créez la classe **`Article`** qui représente un type d'article vendu dans un magasin, par exemple : stylo, pomme, chocolat, … .
2. La classe dispose de l'attribut **`unitPrice`** qui définit le prix unitaire. Ce prix ne contient pas la TVA (EN : VAT).
3. Définissez l'attribut **`quantity`** qui définit le nombre d'unités achetées.
4. Définissez un constructeur permettant d'initialiser le prix unitaire et la quantité à l'aide de paramètres.
5. Développez la méthode **`getTotalPrice`** qui retourne le prix total à payer. Dans ce prix, la TVA est comprise (le taux de la TVA est de 17%). En plus, si la quantité achetée est supérieure à 20, on accorde une remise (EN : discount) de 10% sur le total.

On suppose que les attributs **`unitPrice`** et **`quantity`** ont toujours des valeurs positives.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void
Structures requises : if (simple)

Exercice 14 : NumberPuzzle

Programmez le jeu suivant dans la classe **`NumberPuzzle`** :

L'ordinateur choisit un nombre secret au hasard et sans l'afficher. L'utilisateur doit trouver le nombre secret par essais successifs. Les essais sont comptés et le nombre d'essais est accessible par la méthode **`getCounter()`**.

Réalisation :

- Au démarrage, le nombre secret est zéro.
- Lors d'un appel de la méthode **`selectSecretNumber(int pN)`** l'ordinateur choisit un nombre secret au hasard entre 1 et **`pN`** (on suppose que **`pN`** est positif). Le compteur des essais est mis à zéro.
- L'utilisateur doit trouver le nombre secret par essais successifs en appelant la méthode **`String guess(...)`**.

  A chaque fois que l'utilisateur a entré un nombre, l'ordinateur répond par l'une des trois réponses :

  - "Your number is too big",
  - "Your number is too small" ou
  - "Well done! You found the secret number at the <counter>th guess".

Amélioration :

Dans la méthode **`guess(...)`**, retourner la terminaison correcte pour l'affichage du compteur : "… 1st guess", "... 2nd guess", "... 3rd guess" , "… th guess". Définissez pour cela une variable **`ending`** qui contiendra l'une des chaînes "st", "nd", "rd" ou "th".

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, random
Structures requises : if (imbriqués)

Exercice 15 : Test Qualification

Reprenez la classe **`Test`** :

Ajoutez la méthode **`setRandomMark`** qui remplit **`mark`** au hasard par une note entre 1 et 60.

Ajoutez la méthode **`getQualification`** à la classe **`Test`** pour qu'elle retourne les qualifications suivantes :

Note	***Qualification***
< 10, strictement inférieure à 10	very poor
[10, 20[, comprise entre 10 (inclus) et 20 (exclus)	poor
[20, 30[, comprise entre 20 (inclus) et 30 (exclus)	insufficient
[30, 40[, comprise entre 30 (inclus) et 40 (exclus)	sufficient
[40, 50[, comprise entre 40 (inclus) et 50 (exclus)	good
>= 50, supérieure ou égale à 50	very good

Créez un devoir et vérifiez le fonctionnement correct pour tous les cas.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, random
Structures requises : if (imbriqués)

Exercice 16 : BodyStatistics II

1. Développez la classe **`BodyStatistics`** qui possède les attributs suivants d'une personne :

   age (en ans) , taille (en cm) , poids (en kg) ,
   sexe (*0 <=> masculin, 1 <=> féminin)*

2. Ajouter un constructeur qui initialise les attributs. On suppose (sans vérification) que les données entrées (âge, taille, etc.) sont des données réalistes.
3. Selon la formule de Broca, le poids normal se calcule de la manière suivante :

   `POIDS (kg) = TAILLE (cm) - 100`

   Développez donc la méthode **`getNormalWeight()`** qui calcule le poids normal d'une personne suivant la formule de Broca !

4. Le poids idéal selon Broca est calculé de façon différente pour hommes et femmes :
   pour hommes : poids idéal = poids normal * 0,9
   pour femmes : poids idéal = poids normal * 0,85
   Développez la méthode **`getIdealWeight()`** qui calcule le poids idéal **pour femmes et pour hommes** suivant la formule de Broca !

On suppose (sans vérification) que les données entrées (âge, taille, etc.) sont des données réalistes.

Attention : Ces formules sont valables uniquement pour des adultes et elles donnent des résultats inadaptés pour des personnes très grandes, très petites, très musclées, etc.

5. L'index de corpulence, encore appelé «Indice de Masse Corporelle», ou en anglais "Body Mass Index" (BMI) donne de meilleurs résultats que les indices de Broca. Il se calcule de la manière suivante :

   `BMI = POIDS (kg) / TAILLE² (m²)`

   Programmez donc la méthode **`getBMI()`** qui détermine le BMI d'une personne !

Informations supplémentaires : Pour vous donner une idée du résultat, voici (à gauche) le tableau d'évaluation pour des personnes adultes. A droite un tableau montrant le poids normal au fil de l'âge. D'autres tableaux d'évaluation peuvent être retrouvés sur Internet.

Situation pondérale	BMI Femme	BMI Homme
Maigreur	<19.1	< 20.7
Poids normal/idéal	19.1 - 25.8	20.7 - 26.4
à la limite du surpoids	25.8 - 27.3	26.4 - 27.8
Surpoids	27.3-32.3	27.8 - 31.1
Obésité	> 32.3	> 31.1

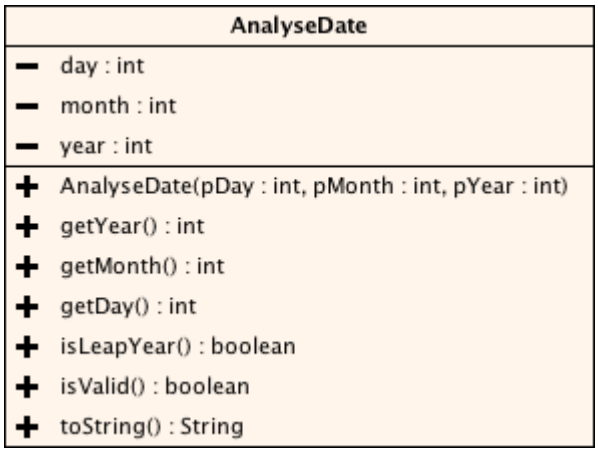
Âge (ans)	Adaptation BMI normal
19–24	-2
25–34	-1
35–44	0
45–54	+1
55–64	+2
> 64	+3

6. Programmez la méthode **`getComment()`** qui retourne une appréciation (situation pondérale) en fonction des tableaux ci-dessus.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, conditions composées
Structure requise : if (imbriqués)

Exercice 17 : AnalyseDate

Implémentez la classe **`AnalyseDate`** de la façon suivante :



La méthode **`isLeapYear`** retourne **`true`**, si l'année donnée est une année bissextile (DE : *Schaltjahr*).

Règles à appliquer pour déterminer si une année est bissextile :

1) Si une année A est un multiple de 4, alors A est une année bissextile.
2) Exception à la règle (1) : Si A est un multiple de 100, alors A n'est pas une année bissextile.
3) Exception à la règle (2) : Si A est un multiple de 400, alors A est une année bissextile.

Programmez la méthode **`isValid`** qui retourne **`true`**, si la date actuelle est une date valable en tenant compte des années bissextiles (p.ex. les dates 29.2.1996 ou 29.2.2000 sont des dates correctes tandis que 21.13.2004, 31.4.2010, 29.2.2003 ou 29.2.1900 sont des dates impossibles).

Recommandation : Il est plus facile d'employer une variable intermédiaire de type booléen pour pouvoir utiliser plusieurs tests simples au lieu d'un seul test très complexe.

La méthode **`toString()`** retourne un texte décrivant la date (p.ex. 12 juin 2011 ou 1er avril 2000).

Notions requises :	classe, objet, méthode, paramètre, type, attribut, void, constructeur, variable booléenne, conditions composées
Structures requises :	if (imbriqués)

Exercice 18 : Equations du second degré

Cet exercice est à résoudre en groupes de 3 à 4 personnes :

Il s'agit d'écrire une classe **`EquationSolver`** qui sert à résoudre des équations de la forme :

$$a \cdot x^2 + b \cdot x + c = 0 \quad (\text{ pour } a,b,c \in \mathbb{R})$$

Indications sur le principe de fonctionnement :

Les données qui définissent l'équation doivent être fournies lors de la création de la classe. Des données pour d'autres équations peuvent être fournies au cours de la vie d'un objet. On peut donc résoudre plusieurs équations avec un même objet.

Après avoir appelé la méthode **`solve()`**, les solutions de l'équation sont mémorisées dans des attributs. La classe possède des méthodes qui savent retourner les différentes solutions sous forme de nombres (type double). Une méthode **`toString()`** retourne comme résultat une représentation de l'équation sous forme de texte. Une méthode **`showSolution()`** affiche toutes les solutions de l'équation actuelle précédées par une ou plusieurs lignes de messages qui indiquent le type de l'équation et le nombre de solutions.

1ère étape :

Discutez en groupe les points suivants :

1. Combien de cas différents faut-il prévoir ? (N'oubliez-pas de prévoir des traitements et des messages pour les cas où a=0). Trouvez des exemples qui pourront vous servir à tester chacun des différents cas possibles.
2. Comment peut-on réagir dans les cas où une équation est indéterminable (infinité de solutions) ou impossible (aucune solution) ?
3. Quels attributs possède votre classe pour mémoriser les données ?
4. Quels attributs possède votre classe pour mémoriser les résultats ?
5. Quelle(s) méthode(s) faut-il prévoir pour fournir les données pour une nouvelle équation ?
6. Quelle(s) méthode(s) faut-il prévoir pour accéder aux différentes solutions de l'équation ?

Notez vos réflexions pour chacun des points précédents.

Etablissez un structogramme pour la méthode **`solve()`**.

Réalisez ensuite la classe selon les spécifications que vous avez fixées. **Commentez** la classe selon le standard ***JavaDoc***. **Testez** la classe pour les différents cas possibles en employant des équations dont vous connaissez les résultats.

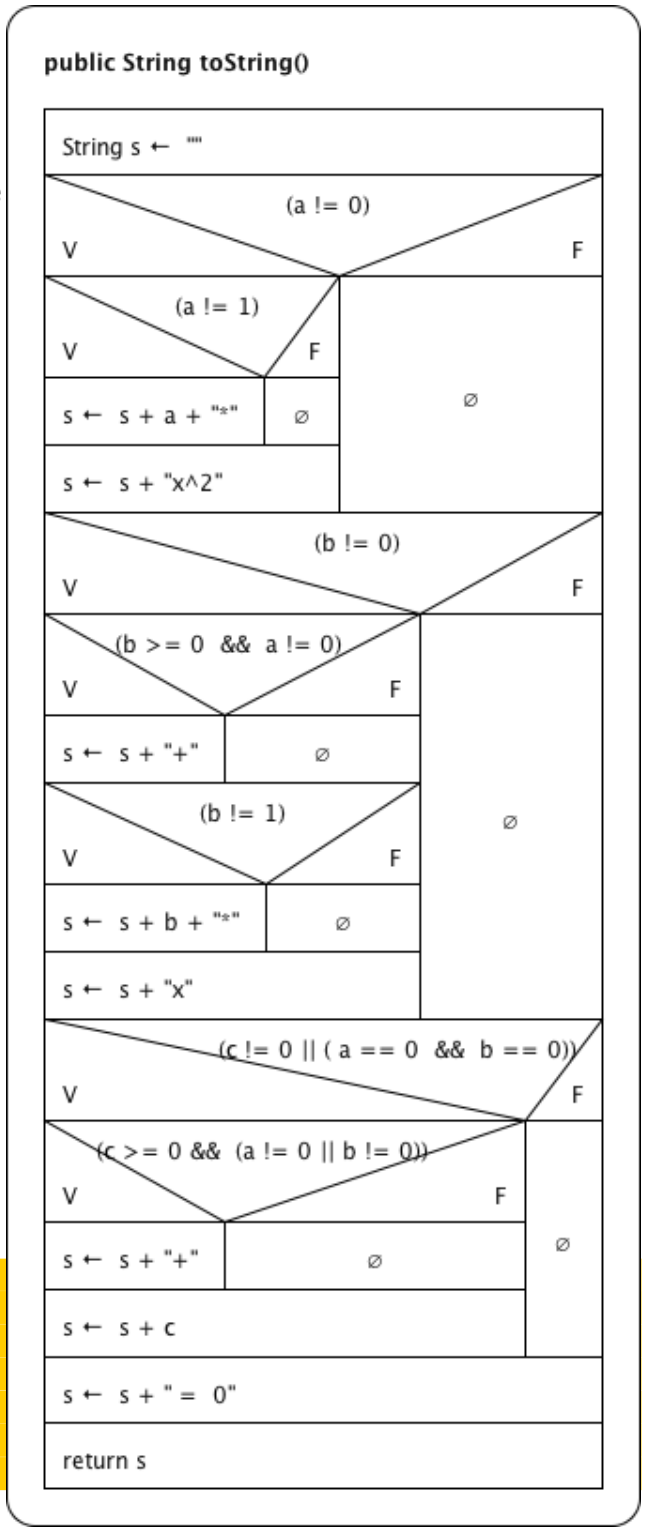
2e étape :

Discutez en groupe les points suivants :

7. Quel problème peut-on rencontrer aussi longtemps que la méthode **`solve()`** n'a pas encore été appelée pour de nouvelles données ? Comment peut-on éviter/résoudre ce problème ?
   Modifiez votre solution en considérant vos conclusions !

8. **Pour avancés :** améliorez la méthode **`toString()`** étape par étape (termes nuls, coefficients égaux à 1, …).

   Pour les pressés : remplacez la méthode **`toString()`** par la méthode décrite par le structogramme ci-contre :



Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur, conditions composées, planification d'une classe
Structures requises : if (imbriqué)

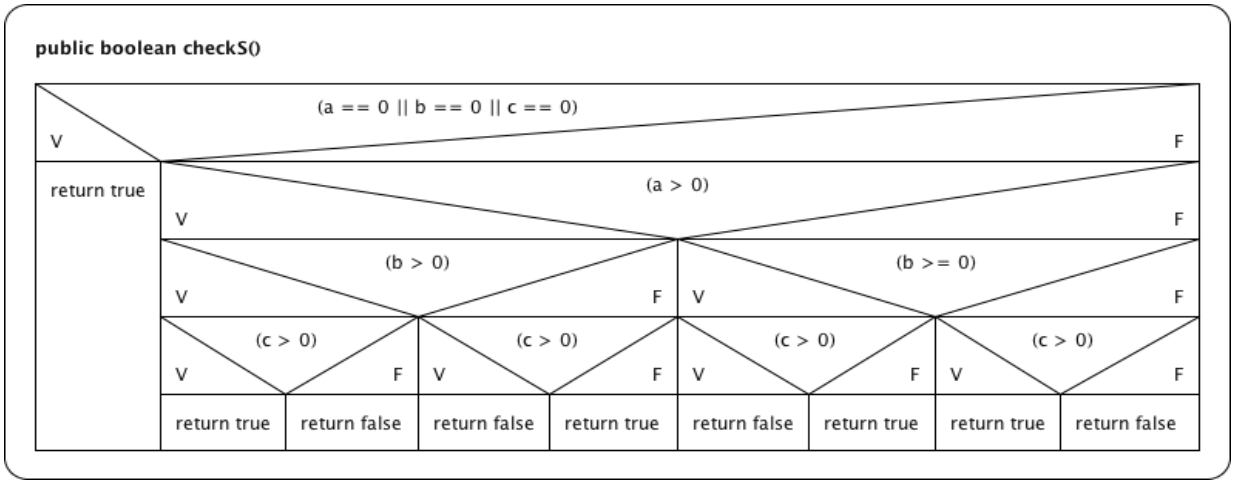
Exercice 19 : Analyse et traduction de structogrammes

Créez une classe **`Structotest`** avec les attributs suivants :

```
double a,b,c;
```

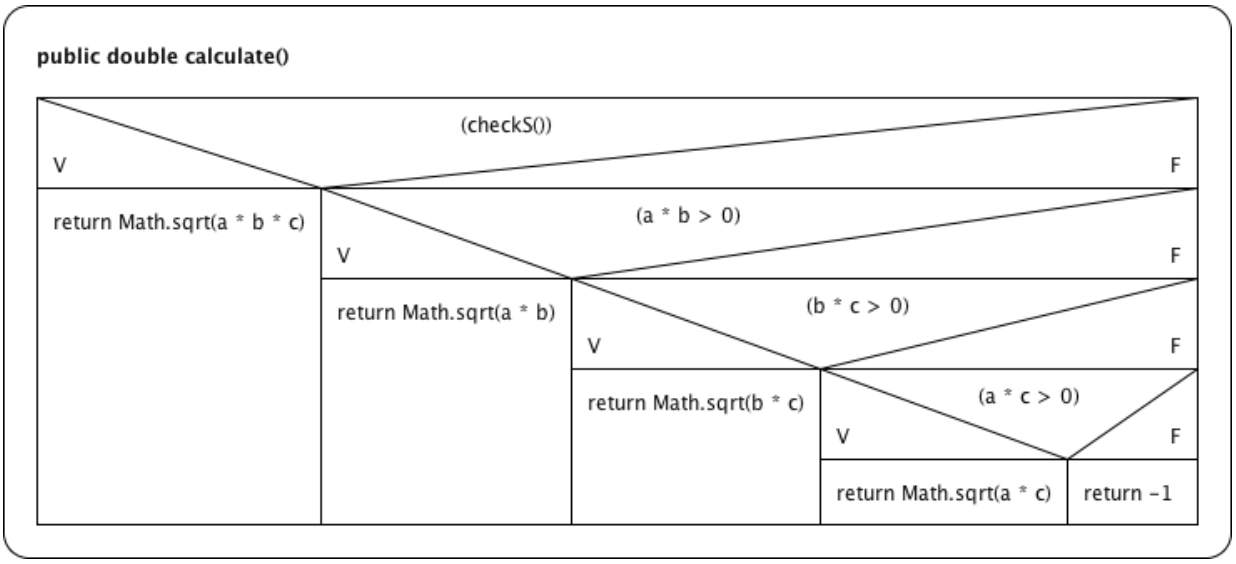
Ajoutez une méthode **`setAll(...)`** qui permet de donner des valeurs aux attributs.

Traduisez le structogramme suivant en Java et intégrez-le dans la classe :



1. Dans quels cas est-ce que la méthode **`checkS()`** retourne **`true`** ?

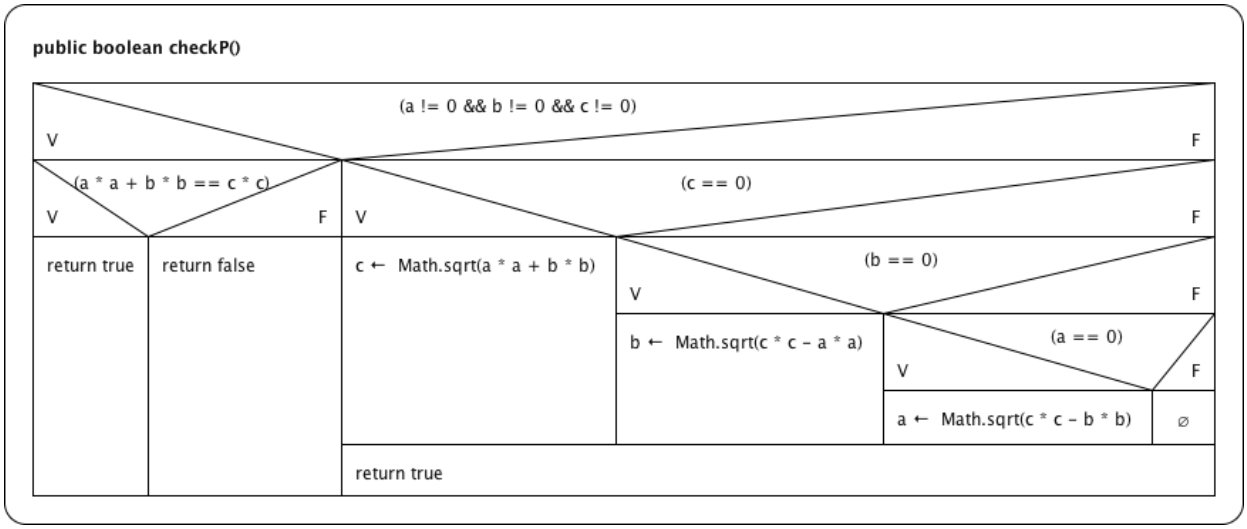
Traduisez le structogramme suivant en Java et intégrez-le dans la classe :



2. Dans quels cas est-ce que cette méthode retourne **`-1`** ?

   Testez vos suppositions avec quelques valeurs !

Traduisez le structogramme suivant en Java et intégrez-le dans la classe :



3. Analysez les différents cas de cette méthode : à quoi sert cette méthode ?
4. Dans quels cas est-ce qu'elle retourne **`true`** ?

   Testez vos suppositions avec quelques valeurs !

Exercice 20 : Calculs avec un entier

Créez la classe **`SimpleCalculationsWithOneInt.`**

La classe dispose de l'attribut **n** qui est un nombre entier positif.

a) Définissez le constructeur **`SimpleCalculationsWithOneInt(int pN)`** qui initialise **n** avec la valeur absolue de **`pN`**.

b) Définissez la méthode **`void printCountUp()`** qui affiche dans la fenêtre des messages tous les nombres entiers de 0 à **n** (par ordre croissant).

c) Définissez la méthode **`void printCountDown()`** qui affiche dans la fenêtre des messages les nombres entiers de **n** à 0 (par ordre décroissant).

d) Définissez la méthode **`int calculateSum()`** qui calcule la somme des **n** premiers nombres entiers.

Exemple : pour **n**=4 le résultat est 1+2+3+4 = 10

e) Définissez la méthode **`double calculateFactorial()`** qui calcule le produit des **n** premiers nombres entiers positifs (non nuls).

Exemple **:** pour **n**=4 le résultat est 1·2·3·4 = 24.0

Remarques :

- En mathématiques, le produit des n premiers nombres est appelé ***factorielle de n*** ou de manière abrégée ***n!***. (angl. : *factorial*, all. : *Fakultät*)
- Par définition : 0! = 1.
- Comme la factorielle devient rapidement très grande, le résultat de la méthode est de type **double**.

Notions requises : classe, objet, méthode, paramètre, type
Structures requises : if, (for || while)

Exercice 21 : Calculs avec un entier (version alternative)

Créez la classe **`SimpleCalculationsWithOneInt_2`** qui effectue exactement les mêmes calculs que **`SimpleCalculationsWithOneInt`** mais en employant l'autre structure répétitive vue dans le cours. C.-à-d. employez maintenant **`while`** si vous avez utilisé **`for`** dans la première version et vice-versa.

Notions requises : classe, objet, méthode, paramètre, type
Structures requises : if, (for || while)

Exercice 22 : Calculs avec deux entiers

Créez la classe **`SimpleCalculationsWithTwoInts`**. La classe dispose des attributs **`a`** et **`b`** qui sont des nombres entiers strictement positifs.

a) Définissez le constructeur **`SimpleCalculationsWithTwoInts(int pA, int pB)`** qui initialise les attributs **`a`** et **`b`** par les valeurs absolues de **`pA`** et **`pB`**.

b) Définissez la méthode **`void printAll()`** qui affiche tous les entiers compris entre **`a`** et **`b`** (**`a`** et **`b`** inclus).

c) Définissez la méthode **`int calculateSumEven()`** qui calcule et retourne la somme de tous les entiers pairs compris entre **`a`** et **`b`** (**`a`** et **`b`** inclus).

d) Définissez la méthode **`double calculatePower()`** qui calcule et retourne la puissance $\mathbf{a^b}$ à l'aide de multiplications successives (donc : ne pas utiliser Math.power() !)

e) Ajoutez la méthode **`swap`** qui échange le contenu des attributs **`a`** et **`b`**. Cette opération s'appelle aussi une *permutation* de deux valeurs.

Les trois méthodes suivantes calculent et retournent le **PGCD** (plus grand commun diviseur) de deux nombres a et b. Chacune d'elles utilise un principe différent.

f) **`int GCD_search()`**
Calcul du PGCD par essais successifs, en commençant par le plus petit des deux nombres a et b.

g) **`int GCD_Euclid()`**
Utilisez l'algorithme d'Euclide par soustractions (→ voir Wikipedia)
Notez la table d'exécution pour a=64 et b=240

h) **`int GCD_OptimizedEuclid()`**
Utilisez l'algorithme d'Euclide par division et reste (→ voir Wikipedia)
Notez la table d'exécution pour a=64 et b=240

Les deux méthodes suivantes calculent et retournent le **PPCM** (plus petit commun multiple de deux nombres a et b.

i) **`int LCM_search()`**
Calcul du PPCM par essais successifs, en commençant par le plus grand des deux nombres a et b.

j) **`int LCM_shortcut()`**
Calcul du PPCM en profitant d'une relation simple qui existe entre a, b, le PPCM et le PGCD de a et b. Pour trouver cette relation, notez les facteurs premiers de deux nombres, de leur PGCD et de leur PPCM (p.ex de a=20 et b=30)... Vérifiez votre théorie pour d'autres nombres a et b.

Notions requises : classe, objet, méthode, paramètre, type
Structures requises : if, (for || while)

Exercice 23 : Calculs avec des nombres aléatoires

Créez la classe **`RandomStatistics`** qui sert à générer des suites de valeurs aléatoires entières tout en effectuant une statistique élémentaire sur les valeurs produites.

La classe possède les attributs suivants :

- **`min`** et **`max`** les limites pour les valeurs aléatoires à générer
- **`count`** le nombre de valeurs produites jusqu'ici
- **`lowest`** le plus petit nombre produit jusqu'ici
- **`highest`** le plus grand nombre produit jusqu'ici
- **`sum`** la somme de toutes les valeurs produites jusqu'ici

La classe possède les méthodes suivantes :

- **`RandomStatistics(long pMin, long pMax)`** initialise les limites et tous les autres attributs. Si **`pMin`** est supérieur à **`pMax`**, les valeurs sont échangées automatiquement.
- **`long getNext()`** retourne comme résultat une valeur aléatoire entre min et max (limites incluses) et actualise les statistiques (les autres attributs)
- **`printSeries(long pN)`** affiche dans la fenêtre des messages une série de n nombres aléatoires entiers entre min et max en actualisant les statistiques. La méthode affiche jusqu'à 20 nombres par ligne.
- **`printStatistics()`** affiche un résumé de la statistique actuelle sous forme de texte. La statistique contient aussi la moyenne (réelle) des nombres générés.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur, Math.random(), conversion de type
Structures requises : if, (for || while)

Exercice 24 : Simulation de jets de deux dés

Développez une classe **`DoubleDice`** qui simule des jets de deux dés et effectue une statistique sur le nombre de fois qu'on a obtenu un 'double' (c.-à-d. deux valeurs identiques). Il est possible d'effectuer une multitude d'essais avec un seul appel d'une méthode.

La statistique contient le nombre de jets effectués, le nombre de 'doubles' et le pourcentage des 'doubles'. (Si on effectue un grand nombre d'essais, on obtient ainsi une approximation de la probabilité de jeter un 'double').

Choisissez les attributs et les méthodes appropriés.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur, Math.random(), conversion de type
Structures requises : if, (for || while)

Exercice 25 : Roulette

Développez une classe **`Roulette`** qui simule des jets de roulette et effectue une statistique sur le nombre de fois qu'on a obtenu un zéro, un nombre pair, un nombre impair, un nombre du groupe 1 (entre 1 et 12), du groupe 2 (entre 13 et 24) du groupe 3 (entre 25 et 36). Il est possible d'effectuer une multitude d'essais avec un seul appel d'une méthode.

La statistique contient le nombre de jets effectués et toutes les autres informations en nombres et en pourcentages.

Choisissez les attributs et les méthodes appropriés.

Notions requises :	classe, objet, méthode, paramètre, type, attribut, void, constructeur, Math.random(), conversion de type
Structures requises :	if, (for || while)

Exercice 26 : IntegerNumber - Caractéristiques d'un entier

Écrivez la classe **`IntegerNumber`** qui possède un attribut **`number`** (un entier positif ou zéro) qui peut être analysé par différentes méthodes. La classe contient les méthodes suivantes :

`IntegerNumber(int pNumber)`	constructeur (initialise **`number`** par la valeur absolue de **`pNumber`**)
`int getNumber()`	accesseur
`boolean isEven()`	retourne *true* ssi le nombre est pair / divisible par deux
`boolean isPrime()`	retourne *true* ssi le nombre est un nombre premier / possède exactement deux diviseurs (1 et soi même)
`int sumOfDividers()`	retourne la somme des diviseurs du nombre (1 et le nombre lui même sont aussi des diviseurs !)
`boolean isPerfect()`	retourne *true* ssi le nombre est parfait : c.-à-d. la somme des diviseurs est égale au double du nombre (Exemples : 6, 28, 496, 8128, ...)
`boolean isDeficient()`	retourne *true* ssi le nombre est déficient : c.-à-d. la somme des diviseurs est strictement inférieure au double du nombre
`boolean isAbundant()`	retourne *true* ssi le nombre est abondant : c.-à-d. la somme des diviseurs est strictement supérieure au double du nombre
`boolean isFriendlyTo(int pOtherNumber)`	retourne *true* ssi **`number`** et **`pOtherNumber`** sont amicaux , c.-à-d. les sommes des diviseurs de **`number`** et de **`pOtherNumber`** sont identiques
`int reverse()`	renverse l'ordre des chiffres dans **`number`** et retourne le résultat, p.ex.: si avant l'appel de reverse() **`number`** est égal à 1234, alors après l'appel, **`number`** sera 4321.
`boolean isPalindrome()`	retourne *true* si le nombre est un palindrome : c.-à-d. le nombre est le même si on le lit de gauche à droite ou de droite à gauche (p.ex. : 13531)

Notions requises :	classe, objet, méthode, paramètre, type
Structures requises :	if, (for || while)

Exercice 27 : Fractions

Programmez la classe **`Fraction`** qui représente une fraction mathématique !

La classe doit avoir le constructeur que voici :

- **`Fraction(int pNumerator, int pDenominator)`**

On suppose (sans vérification) que le dénominateur est différent de zéro, sinon les résultats des calculs sont imprévisibles.

Cette classe doit avoir les méthodes suivantes :

- **`int getNumerator()`**
- **`int getDenominator()`**
- **`double getDecimal()`** retourne la valeur décimale de la fraction
- **`String toString()`**
  retourne un texte de la forme : *numérateur/dénominateur (val.décimale)*
  si le dénominateur est 1, alors il n'est pas affiché
  Exemples : **`8/16 (0.5)`**
  `125/100 (1.25)`
  `23 (23.0)`
- **`int gcd()`**
  calcule et retourne le PGCD de a et b (calculé selon la méthode d'Euclide par division et reste).
- **`simplify()`**
  simplifie la fraction en utilisant la méthode ci-dessus.

Les 4 méthodes **`add`**, **`subtract`**, **`multiply`**, **`divide`** (→ voir plus bas) permettent d'effectuer des calculs entre deux objets du type **`Fraction`**. On peut ajouter, soustraire, diviser et multiplier la fraction actuelle à/par une deuxième fraction passée comme paramètre.

La nouveauté pour cet exercice est qu'on peut aussi **passer un objet comme paramètre**, de la même façon qu'on peut passer des nombres ou des textes comme paramètres.

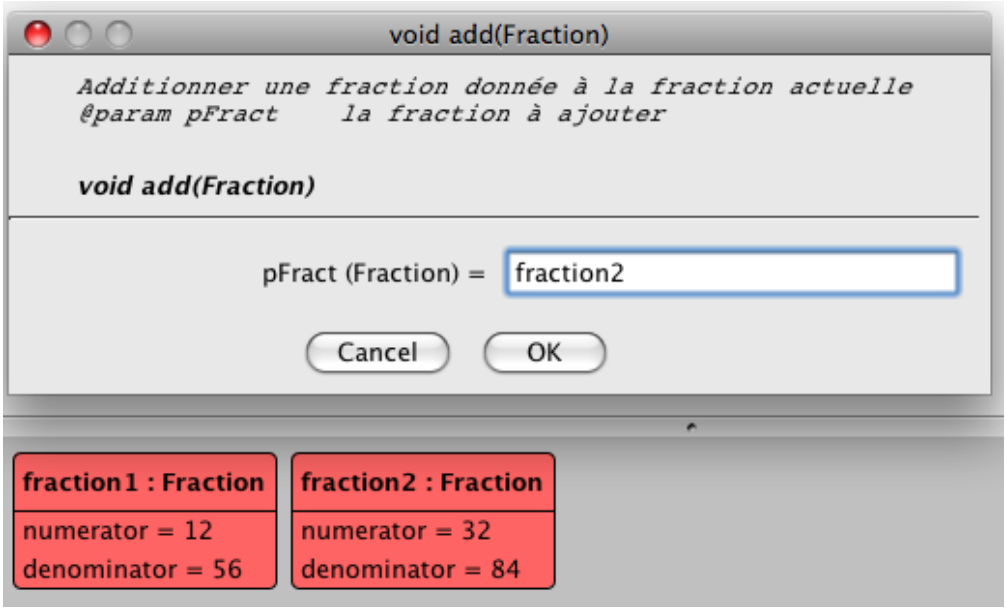
Pour tester ces méthodes en *Unimozer*, il faut que vous ayez créé deux instances de la classe **`Fraction`**.

Ensuite, vous pouvez par exemple ajouter une fraction à l'autre, en passant le nom de la deuxième fraction comme paramètre à la méthode **`add`** de la première fraction.

- **`add(Fraction)`** ajouter et simplifier
- **`subtract(Fraction)`** soustraire et simplifier
- **`multiply(Fraction)`** multiplier et simplifier
- **`divide(Fraction)`** diviser et simplifier

Exemple d'utilisation en Unimozer illustrant l'addition de **fraction2** à **fraction1** :

Ici, l'utilisateur a effectué un clic droit sur **fraction1** puis il a choisi la méthode '**add**'.



Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur
Structures requises : if, (for || while)

Exercice 28 : Comptes bancaires II

La banque *RabloBank* a codé ses numéros de compte de façon à ce qu'il soit possible de contrôler si un numéro de compte a été entré correctement. Pour ce faire, les deux derniers chiffres du numéro correspondent à la somme des chiffres précédents (DE : *Quersumme*). Les numéros de compte (chiffres de contrôle inclus) sont toujours des nombres à 9 chiffres.

Exemples :

123450015	est donné correctement, car	1+2+3+4+5	= 15
110021106	est donné correctement, car	1+1+2+1+1	= 06
643553947	n'est pas donné correctement, car	6+4+3+5+5+3+9	= 35 ≠ 47

Reprenez votre projet **`Account`** et sauvez-le sous **`Account2`**. Modifiez la classe **`Account`** comme suit :

1. Chaque compte possède un numéro de compte **`number`**, qui peut être consulté avec **`getNumber()`**.
2. La méthode **`isValid(long pN)`** contrôle si un numéro de compte donné comme paramètre est valide (selon la description ci-dessus).
3. La méthode **`generateNumber()`** sert à générer un numéro de compte valide de façon aléatoire.
   (On néglige ici le fait qu'en réalité il est impossible d'employer une telle méthode parce qu'elle risque de produire deux fois le même numéro de compte.)
4. Lors de la création d'un compte, il faut vérifier que le numéro de compte donné comme paramètre est valide, sinon un numéro de compte valide doit être créé de façon aléatoire.
5. La méthode **`transfer(...)`** permet de transférer une somme d'argent du compte actuel vers un autre compte. Pour effectuer le transfert, il faut indiquer comme paramètre le compte (objet) destinataire, le numéro de compte destinataire (comme contrôle) et le montant à transférer. Avant d'effectuer le transfert, il faut vérifier,

   * si le numéro de compte destinataire est valide,

   * si le nom du compte destinataire correspond au numéro de compte donné,

   * s'il reste assez d'argent pour effectuer le transfert.

   Comme résultat, la méthode **`transfer(...)`** retourne un texte indiquant le succès de l'opération ou une description de l'erreur commise, p.ex. :

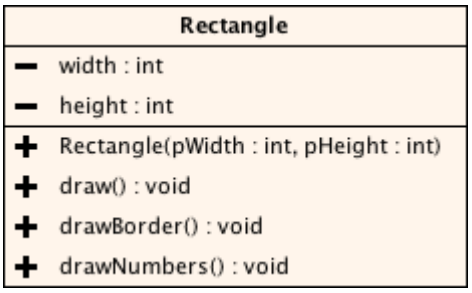
   "ERROR: The remitte's account number is not valid!"

   "SUCCESS: The transfer has been accomplished successfully!"

   …
6. La méthode **`toString()`** retourne un texte contenant le numéro de compte et le solde actuel

Notions requises :	classe, objet, méthode, paramètre, type, attribut, void, constructeur, division entière avec reste
Structures requises :	if, (for || while)

Exercice 29 - Structures répétitives imbriquées



Créez la classe **`Rectangle`** avec les attributs entiers positifs **`width`** (largeur) et **`height`** (hauteur) et définissez un constructeur permettant d'initialiser les attributs (prenez la valeur absolue des données fournies).

a) Définissez la méthode **`void draw()`** qui dessine un rectangle à l'aide d'étoiles.

Exemple pour **`width`**=10 et **`height`**=4 :

```
**********
**********
**********
**********
```

b) Définissez la méthode **`void drawBorder()`** qui dessine un rectangle en dessinant uniquement le bord à l'aide d'étoiles.

Exemple pour **`width`**=10 et **`height`**=4 :

```
**********
*        *
*        *
**********
```

c) Définissez la méthode **`void drawNumbers()`** qui "dessine" un rectangle à l'aide de nombres.

Exemple pour **`width`**=10 et **`height`**=4 :

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur
Structures requises : if, (for || while) imbrication de boucles

Exercice 30 : Figures d'étoiles

Écrivez la classe **`Stars`** qui permet de dessiner différentes figures en employant le caractère '*'. (P.ex. en 'dessinant' 4 lignes contenant chacune 4 étoiles, on obtient un carré de côté 4.). On suppose que toutes les données fournies comme paramètres sont positives.

Notions requises : classe, objet, méthode, paramètre, type, attribut, void, constructeur
Structures requises : if, (for || while) imbrication de boucles

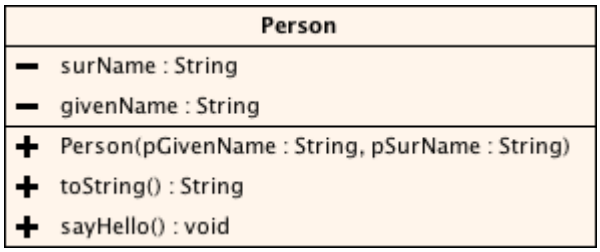
Exercice 31 : Comptes bancaires III - MiniBay

Idée : Simulation de l'achat et de la vente d'articles. Modélisation à développer en groupes ou en discussion ensemble avec la classe.

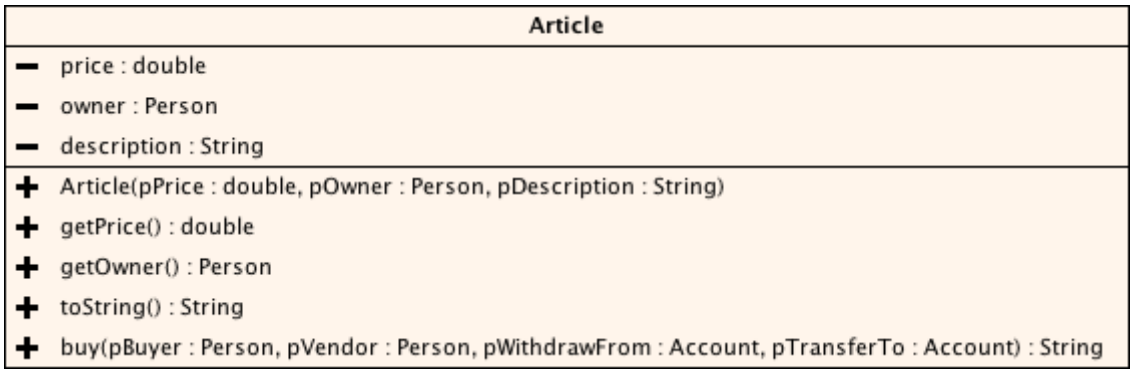
Exemple de réalisation :

Reprenez votre projet **`Account2`** et sauvez-le sous **`MiniBay`**.

Ajoutez la classe **`Person`** :



Ajoutez une classe **`Article`** :



La méthode **`buy`** permet de réaliser l'achat d'un article, tout en vérifiant que toutes les données sont exactes. Dans tous les cas un message résumant l'action est retourné comme résultat.

Pour mémoriser le titulaire d'un compte, ajoutez un attribut **`holder`** (classe **`Person`**) et un accesseur **`getHolder()`** à la classe **`Account.`**

Créez plusieurs personnes et créez un compte pour chacune des personnes. Chaque compte est initialisé avec une somme de départ.

Créez plusieurs articles et attribuez-les aux personnes.

Simulez les achats d'articles.

Notions requises :	classe, objet, méthode, paramètre, type, attribut, void, constructeur, interactions entre objets
Structures requises :	if

Exercice 32 : Jeu - deviner des nombres

Reprendre le jeu **NumberPuzzle** et sauvez-le projet sous un nouveau nom.

Ajouter à la classe **NumberPuzzle** un attribut **display** du type booléen et les méthodes **enableDisplay()** et **disableDisplay()** qui changent sa valeur. Au démarrage, **display** est désactivé.

Modifiez la méthode **guess()** comme suit : Si **display** est activé, alors à chaque essai, la valeur de l'essai ainsi que la réponse et le nombre d'essais sont affichés dans la fenêtre console.

Modifiez la méthode **selectSecretNumber()** comme suit : Si **display** est activé, alors un trait de séparation est affiché dans la fenêtre console.

Programmez une classe **Player** qui permet de jouer au jeu de la devinette en essayant de trouver la valeur cachée.

- la méthode **int dummy(NumberPuzzle,int)** essaie les valeurs l'une après l'autre,
- la méthode **int smarty(NumberPuzzle,int)** essaie de trouver la valeur secrète en éliminant après chaque essai la moitié des valeurs restantes.

Les deux méthodes retournent le nombre d'essais comme résultat.

Exemple de l'exécution de la méthode **smarty** (attribut **display** activé) :

```
================================================================================
Guess nr. 1: 50
Your number is too small.
Guess nr. 2: 75
Your number is too big.
Guess nr. 3: 62
Your number is too small.
Guess nr. 4: 68
Your number is too big.
Guess nr. 5: 65
Your number is too big.
Guess nr. 6: 63
Your number is too small.
Guess nr. 7: 64
Well done! You found the secret number at the 7th guess.
```

Pour avancés :

Développez la classe **Player2** qui est identique à **Player**, mais qui contient un attribut du type **NumberPuzzle** qui est initialisé comme suit :

```
protected NumberPuzzle game = new NumberPuzzle();
```

Dans cette classe, les paramètres du type **NumberPuzzle** ne sont plus nécessaires.

Notions requises :	classe, objet, méthode, paramètre, type, attribut, void, constructeur, interaction entre objets
Structures requises :	if, (for || while)

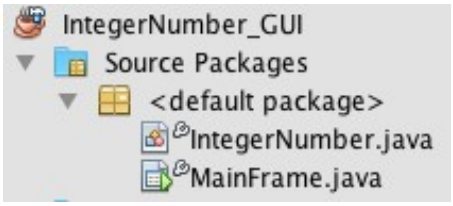
Exercice 33 : IntegerNumber – GUI

Dans cet exercice, nous allons intégrer la classe **IntegerNumber** (de l'exercice 27) dans un programme autonome avec une interface graphique. Ainsi, vous obtenez un programme indépendant qui peut vous afficher simplement toutes les caractéristiques des nombres entrés.

Ouvrez pour cela le programme *NetBeans* sur votre machine. (Ne vous laissez pas impressionner par la 'complexité' apparente de ce programme. Vous n'aurez besoin que de quelques fonctions de ce programme et vous serez guidés étape par étape. L'année prochaine vous allez utiliser ce programme pour développer vous mêmes des interfaces graphiques.)

Pour l'instant, vous allez seulement ajouter votre classe de l'exercice 27 au projet NetBeans qui contient l'interface déjà préfabriquée :

1. Ouvrez d'abord votre projet **IntegerNumber** en NetBeans, avec le menu : **File-OpenProject...** (Vous allez constater, qu'un projet Unimozer se laisse ouvrir parfaitement en NetBeans. Il est cependant impossible de créer des objets par un clic de la souris ou de tester les méthodes comme en Unimozer.)
2. Pour vérifier, vous pouvez ouvrir le fichier **IntegerNumber.java** par un double-clic. Vous y retrouvez le même code qu'en Unimozer.
3. Ouvrez maintenant en NetBeans le projet **IntegerNumber_GUI** (que votre professeur vous a fourni ou que vous avez téléchargé du site **java.cnpi.lu**).
4. Ouvrez le fichier **MainFrame.java** par un double-clic. Cette classe contient l'interface graphique du programme. Vous pouvez changer entre les vues '*Design*' et '*Source*'.
5. Dans la vue '*Source*', vous constatez qu'une ligne est marquée et contient un texte souligné en rouge. Si vous passez avec la souris au-dessus du symbole  ou du texte souligné, vous trouvez le message **"cannot find symbol – class IntegerNumber"**. Ceci est bien logique puisque le projet **IntegerNumber_GUI** ne contient pas et ne connaît pas la classe **IntegerNumber.java**.
6. Copiez donc votre classe **IntegerNumber.java** dans le dossier <default package> du projet **IntegerNumber_GUI**. Pour cela, cliquez avec le bouton droit sur la classe **IntegerNumber.java** et choisissez **'Copy'**. Cliquez ensuite avec le bouton droit de la souris sur <default package> du projet **IntegerNumber_GUI** et choisissez **Paste → Copy**. Maintenant le projet **IntegerNumber_GUI** devrait se présenter comme suit :



7. Pour éviter des confusions entre les deux classes **IntegerNumber.java**, vous pouvez maintenant fermer votre projet **IntegerNumber** (clic droit sur le projet, puis choisissez **Close**). Laissez le projet **InternetNumber_GUI** ouvert.
8. Si tout s'est bien passé, dans le code source de **MainFrame.java**, le symbole et le message d'erreur devraient avoir disparu. (Sinon, vérifiez que votre classe **IntegerNumber** corresponde exactement aux descriptions de l'exercice 27 et adaptez votre classe. Vérifiez surtout les noms de la classe et des méthodes dans

IntegerNumber.java)

9. S'il ne reste plus d'erreurs ni d'incompatibilités, vous pouvez faire tourner le programme **IntegerNumber_GUI** (clic droit, puis **Run** ou cliquez sur le triangle vert dans la barre des outils).
10. Vous pouvez simplement entrer un nombre dans la boîte texte, puis pousser sur *Enter*. Toutes les caractéristiques du nombre entré sont affichées en même temps dans les champs prévus.

Ouvrez la classe **MainFrame.java** dans le mode '*Source*'. Considérez la méthode **inputTextFieldActionPerformed**. C'est cette méthode qui est exécutée automatiquement lorsque vous poussez sur *Enter* dans le champ texte.

Analysez le code de cette méthode pour comprendre le fonctionnement du programme, p.ex :

a) Que fait la première ligne de la méthode **inputTextFieldActionPerformed** ? Quelle est l'importance de cette ligne ?

b) Que représente la variable **number** ?

c) Quel est le rôle du mot clé '**new**' ?

d) Que fait la méthode **Integer.valueOf** ?

e) Comment fonctionne l'affichage dans les libellés ?

f) . . .

Pour avancés : Obtenir un programme indépendant

- Compilez le programme **IntegerNumber_GUI** : clic droit, puis "**Clean and Build**".
- Fermez NetBeans.
- Trouvez le projet **IntegerNumber_GUI** sur votre disque dur.
- Ouvrez le dossier **dist** dans ce projet.
- Faites un double-clic sur **IntegerNumber_GUI.jar**.

De cette façon, vous obtenez un programme Java qui sait tourner indépendamment de NetBeans ou Unimozer sur toute machine qui sait exécuter Java (et ceci sur Windows, Mac ou Linux).

Pour utiliser le programme sur une autre machine, il suffit d'y copier le fichier **IntegerNumber_GUI.jar** et de le lancer.

Exercice 34 : JackOnDice (avec GUI)

Dans cet exercice, il s'agit de développer un petit jeu qui sera ensuite intégré dans une interface graphique donnée. Vous aurez besoin du fichier **JackOnDice.jar** et du projet **JackOnDice_GUI** que votre professeur vous a fournis ou que vous pouvez télécharger du site **java.cnpi.lu**.

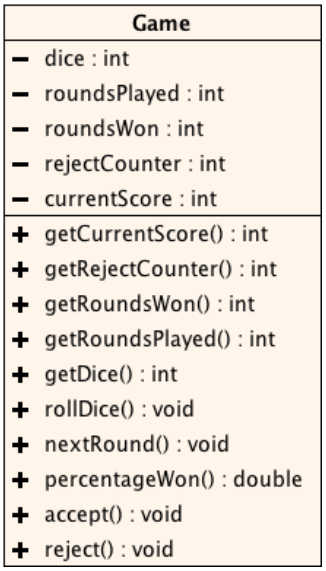
Dans ce jeu, il faut essayer d'atteindre exactement 21 points avec des jets de dé *(all. : Würfel, angl. dice)* successifs. A chaque jet de dé, plusieurs options sont possibles :

- ***'accept' :*** Il faut cliquer sur ***accept*** pour ajouter le nombre des yeux du dé au score. Si on a obtenu exactement 21 points, on a gagné la partie et on passe automatiquement à la prochaine partie. Si on a dépassé les 21 points, on a perdu la partie et on passe automatiquement à la prochaine partie.
- ***'reject' :*** Il est cependant possible de relancer le dé sans ajouter les yeux au score, (p.ex. si on dépasserait les 21 points). Ceci peut se faire au maximum 2 fois par partie. Si on relance une troisième fois, on a perdu la partie et on passe automatiquement à la prochaine partie.
- ***'Give up / next round' :*** à n'importe quel moment, on peut abandonner la partie, si on voit qu'on n'atteindra pas exactement les 21 points. Dans ce cas la partie est perdue et on passe automatiquement à la prochaine partie.

→ Ouvrez maintenant (par un double-clic) le programme **JackOnDice.jar**. Il s'agit de la version finie du jeu. Jouez quelques parties pour voir comment le programme fonctionne ...

Créez en Unimozer un nouveau projet **JackOnDice** et créez-y la classe **Game.java**.

1. Créez les attributs et les accesseurs (**get...**) selon le diagramme UML. Tous les attributs sont initialisés à zéro.



Ajoutez les méthodes suivantes :

2. La méthode **rollDice()** lance le dé, c.-à-d. elle affecte une valeur aléatoire entre 1 et 6 à **dice**.
3. La méthode **nextRound()** passe à la prochaine partie de jeu, c.-à-d. : elle lance le dé, augmente le nombre de parties jouées et remet le score actuel ainsi que le compteur **rejectCounter** à zéro.
4. La méthode **accept()** ajoute les yeux du dé au score, puis elle réagit aux 3 cas suivants :
   a) on a atteint le score de 21 : la partie est gagnée et on passe à la partie suivante,
   b) on a dépassé le score de 21 : la partie est perdue et on passe à la partie suivante,
   c) le score n'a pas dépassé 21 : le dé est relancé.
5. La méthode **reject()** est appelée si on ne veut pas ajouter le nombre donné. Cette méthode augmente le compteur des relancements (**rejectCounter**), puis réagit aux cas suivants :
   a) on a dépassé la limite de deux relancements : la partie est perdue et on passe à la partie suivante,
   b) on n'a pas dépassé la limite de deux relancements : on relance le dé.
6. La méthode **percentageWon()** calcule et retourne comme nombre réel le pourcentage de parties gagnées.

Lors de la réalisation des méthodes ci-dessus, limitez la duplication du code à un minimum, c.-à-d. faites appel aux méthodes *rollDice()* et *nextRound()* partout où c'est possible (SANS COPIER le code qui s'y trouve) !

Testez le bon fonctionnement des méthodes de la classe en Unimozer.

Lorsque tout fonctionne correctement, intégrez la classe **Game.java** dans le projet NetBeans **JackOnDice_GUI**.

Pour ce faire, procédez de la même façon que dans l'exercice précédent, c.-à-d. :

1. Ouvrez les projets **JackOnDice** et **JackOnDice_GUI** en NetBeans.
2. Copiez la classe **Game.java** de **JackOnDice** vers **JackOnDice_GUI**.
3. Fermez le projet **JackOnDice**.
4. Vérifiez qu'il ne reste pas d'incompatibilités et corrigez-les si nécessaire.
5. Faites tourner le programme :-)

Ouvrez ensuite la classe **MainFrame.java** en mode '*Source*'.

Analysez le code de cette classe pour comprendre le fonctionnement du programme, p.ex :

g) Que fait la première ligne de la classe? Quelle est l'importance de cette ligne ?

h) Que représente l'attribut **game**?

i) Quel est le rôle du mot clé '**new**' ?

j) Quelles méthodes sont exécutées lorsqu'on clique sur les différents boutons ? Que font-elles ?

k) Quel est le rôle de la méthode **updateView()** ? Pourquoi est-ce que le programmeur a décidé de créer cette méthode ?

l) Que fait la méthode **String.valueOf** ?

m) Quelles actions sont effectuées au démarrage ? Où sont définies ces actions ?

n) Que remarquez-vous en ce qui concerne les noms des composants utilisés dans l'interface graphique ?

o) ...

Pour avancés : Obtenir un programme indépendant

Evidemment, le jeu **JackOnDice** peut aussi être compilé et lancé indépendamment de NetBeans…

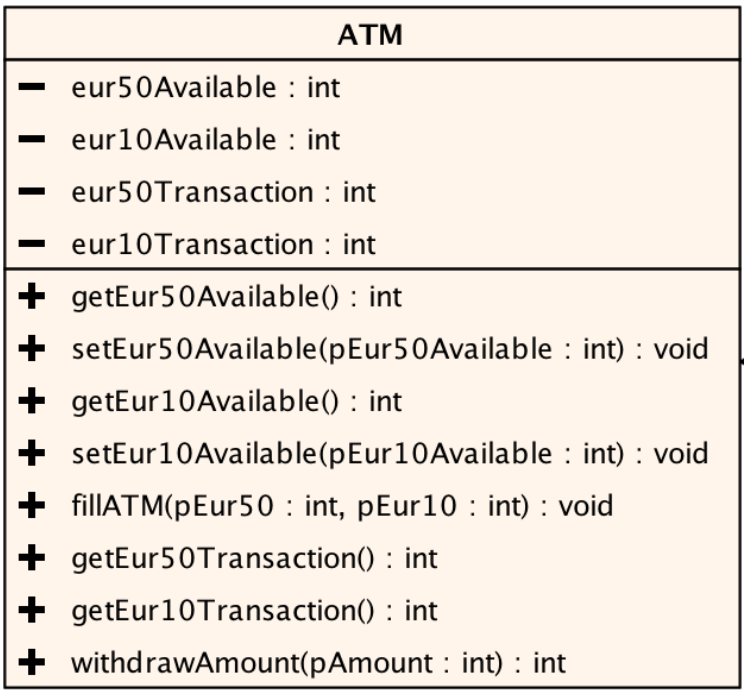
Exercice 35 : ATM (avec GUI)

Dans cet exercice, il s'agit de réaliser un distributeur de billets de banque (angl. **A**utomatic **T**eller **M**achine), appelé au Luxembourg "*Bancomat*".

Le programme sera ensuite intégré dans une interface graphique donnée. Vous aurez besoin du fichier **ATM.jar** et du projet **ATM_GUI** que votre professeur vous a fournis ou que vous pouvez télécharger du site **java.cnpi.lu**.

Créez en Unimozer le projet **ATM** et ajoutez-y la classe **ATM.java**.

La classe **ATM.java** contient la logique de fonctionnement de l'ATM.



Créez les attributs décrit ci-dessous ainsi que les accesseurs et manipulateurs représentés dans le diagramme UML. Tous les attribut sont initialisés à zéro.

eur50Available Nombre de billets de 50 euros disponibles dans l'ATM
eur10Available Nombre de billets de 10 euros disponibles dans l'ATM
eur50Transaction Nombre de billets de 50 euros distribués après une demande de retrait
eur10Transaction Nombre de billets de 10 euros distribués après une demande de retrait

La méthode **fillATM** permet de réinitialiser **eur50Available** et **eur10Available** aux valeurs données dans les deux paramètres.

La méthode **withdrawAmount** permet d'effectuer un retrait de billets. Elle contient du code qui n'est pas correct mais qui vous permet de réaliser et de tester l'étape 2 de cet exercice si vous ne parvenez pas à réaliser le développement demandé ci-après. Mettez ce code en commentaire.

Réalisez la méthode **withdrawAmount** de manière à calculer le nombre de billets de 50 et 10 euros à distribuer pour un montant demandé et à mettre à jour le nombre de billets disponibles dans l'ATM.

Exemple:

Nombre de billets disponibles dans l'ATM avant le retrait: 10 x 50€ , 10 x 10€

Montant demandé: 130€

Nombre de billets à distribuer: 2 x 50€ , 3 x 10€

Nombre de billets disponibles dans l'ATM après le retrait: 8 x 50€ , 7 x 10€

La méthode **withdrawAmount** retourne un code (valeur numérique) qui indique si l'action a pu être effectuée avec succès.

- S'il y a suffisamment de billets de 50€ et de 10€ disponibles dans l'ATM alors la transaction réussit, la méthode retourne la valeur 0.
- S'il n'y a pas suffisamment de billets dans l'ATM, la méthode retourne la valeur 2.
- Si le montant demandé est invalide, car ce n'est pas un multiple de 10 (p.ex. 35€), alors la méthode retourne la valeur 1.

Testez en Unimozer le bon fonctionnement des méthodes de la classe.

Lorsque tout fonctionne correctement, intégrez la classe **ATM.java** dans le projet NetBeans **ATM_GUI**. Pour ce faire, procédez de la même façon que dans les deux exercices précédents. Démarrez ensuite le programme.

Ouvrez ensuite la classe **MainFrame.java** dans le mode '*Source*'.

Analysez le code de cette classe pour comprendre le fonctionnement du programme. P.ex :

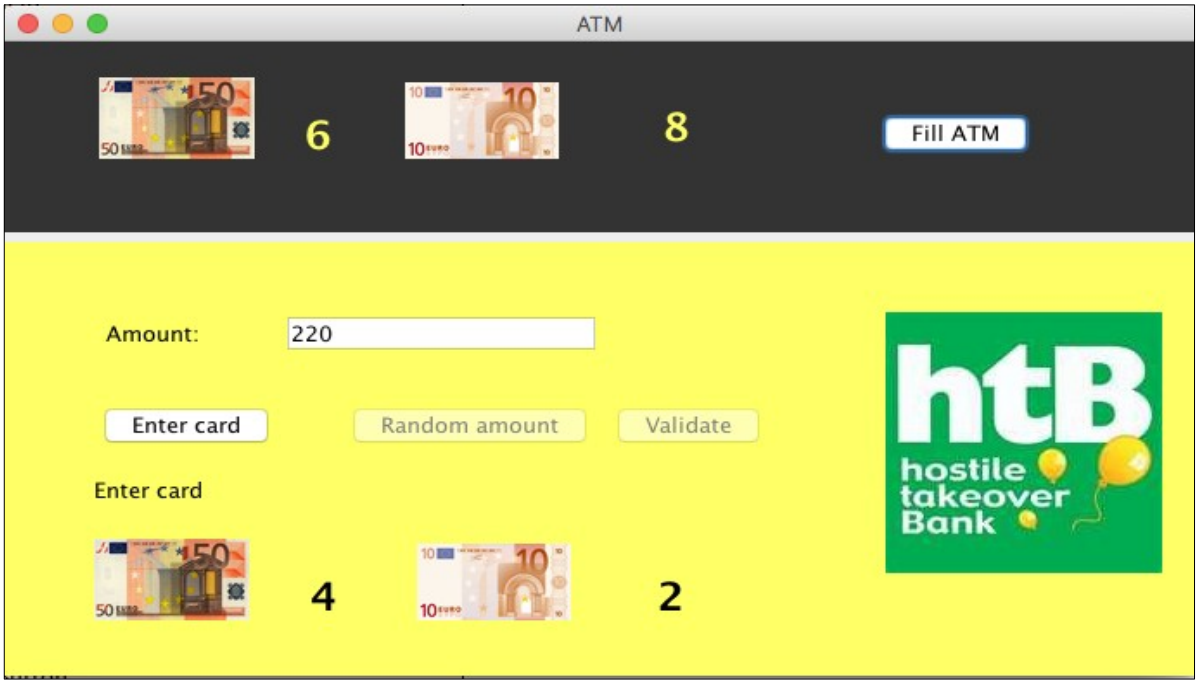
a) Que se passe-t-il si on met des commentaires '//' devant la définition de l'attribut **atm**? Comment exactement est-ce que NetBeans nous signale l'erreur commise ?

b) Analysez et décrivez les actions effectuées par les différentes lignes des méthodes dont le nom se termine par '**ActionPerformed**'.

c) A quoi sert finalement le code retourné par la méthode **withdrawAmount** ? Où et comment est-il considéré ?

d) Pourquoi est-ce que le programmeur a décidé de définir une variable '**response**' ?

e) …


ATM
6
8
Fill ATM
Amount:
220
Enter card
Random amount
Validate
Enter card
4
2
htB
hostile
takeover
Bank

Exercice 36 : Développement d'un jeu en GreenFoot

Greenfoot est un environnement qui permet de façon très simple de développer des jeux amusants en Java avec une interface graphique intéressante et qui permet d'interagir avec les objets visuels. Le fait que le développement est basé sur du code Java vous permet de programmer de façon beaucoup plus flexible des programmes plus intéressants que dans des environnement renfermés (comme p.ex. Scratch).

L'environnement est gratuit et disponible pour tous les systèmes usuels.

Ouvrez le site : https://www.greenfoot.org

Sur la page 'Activity', vous pouvez des projets de différentes qualités et complexités qui ont été développés par la communauté *Greenfoot*. Vous pouvez les tester dans votre 'Browser'.

Pour pouvoir développer vos propres projets, il faut d'abord installer l'environnement *Greenfoot* : https://www.greenfoot.org/download

Sur votre ordinateur personnel à la maison, vous pouvez simplement installer la version qui correspond à votre ordinateur (Windows, Mac, Linux/Ubuntu). A l'école, vous n'avez probablement pas la possibilité d'installer des logiciels supplémentaires (il vous manque les droits d'administrateurs). Vous pouvez cependant simplement installer et démarrer la version '*Standalone*' sur une clé USB. Une autre possibilité est d'employer la version 'Pure Java'. Pour cela, vous devez savoir où se trouve le JDK *(Java Development Kit)* sur votre ordinateur. Votre professeur vous aidera à trouver la meilleure méthode.

Avant de vous lancer dans votre projet de jeu personnel, il est conseillé de suivre d'abord le tutoriel à 6 petites étapes que vous trouvez sous '*Documentation*' :

https://www.greenfoot.org/doc/tut-1

Ce tutoriel vous explique à travers un jeu (avec des crabes) les bases du développement en *Greenfoot*. Au besoin, des notions avancées sont expliquées dans d'autres tutoriels.

Il existe aussi une série de tutoriels sous forme de vidéos en anglais et en allemand :

https://www.greenfoot.org/doc/joy-of-code

https://www.youtube.com/playlist?list=PLLTmbYoiafl8BHRr2OwAsBZF9KLgf9Xvc

Réalisez ensuite un petit jeu de votre choix

selon votre goût personnel !

Conseil :

Développez vos jeux en commençant par une base simple et continuez en les faisant évoluer étape par étape. Testez et 'déboguez' chaque version intermédiaire.

En développant par des améliorations successives, vous aurez toujours des expériences positives au fur et à mesure que vous avancez et les défis que vous rencontrerez resteront surmontables et facilement gérables.

Exercice 37 : TurtleBox (GUI)

Le projet **TurtleBox** vous permet de diriger une tortue sur une surface de dessin. La tortue possède un 'crayon' à l'aide duquel elle peut faire des dessins en se déplaçant.

Ainsi, vous pourrez par du code Java et en déplaçant la tortue faire des dessins à l'écran. Tout ceci se fait en appelant les méthodes publiques de la classe **TurtleBox** que vous pouvez simplement ouvrir en Unimozer.

Ouvrez le projet existant **`TurtleBox`** en Unimozer. Ce projet peut être téléchargé à l'adresse suivante : http://unimozer.fisch.lu/Files/TurtleBox.zip ou sur **java.cnpi.lu**.

Vous y trouvez la classe **`TurtleBox`** :

TurtleBox
− LINE : int
− MOVE : int
− surface : JPanel
− turtlePos : Point2D.Double
− turtleHidden : boolean
− penIsDown : boolean
− angle : double
− speed : int
− elements : Vector<Vector<Integer>>
+ TurtleBox(width : int, height : int)
− drawThickLine(g : Graphics, x1 : double, y1 : double, x2 : double, y2 : double, thickness : int, c : Color) : void
− drawTurtle(graphics : Graphics2D, innerColor : Color, outerColor : Color) : void
+ paint(graphics : Graphics) : void
− getAngle() : double
− setAngle(angle : double) : void
+ setSpeed(pSpeed : int) : void
+ left(angle : double) : void
+ right(angle : double) : void
+ hideTurtle() : void
+ showTurtle() : void
+ penUp() : void
+ penDown() : void
− addElement(kind : int, x1 : double, y1 : double, x2 : double, y2 : double) : void
+ forward(pixels : double) : void
+ backward(pixels : double) : void
− gotoXY(x : double, y : double) : void
+ clear() : void
+ home() : void
+ main(args : String[]) : void

Vous pouvez simplement améliorer les capacités de la tortue en ajoutant des méthodes qui font appel aux méthodes de base...

Lors des dessins, les angles sont indiqués en degrés et toutes les positions et coordonnées sont exprimées en points à l'écran.

Effectuez les tâches suivantes :

- Compilez le projet et créez une instance de cette classe !
- Utilisez les méthodes **`forward(...)`**, **`backward(...)`**, **`right(...)`** et **`left(...)`** pour faire avancer la tortue de manière à ce qu'elle dessine les éléments que voici :
  - un carré
  - un triangle équilatéral
  - un triangle rectangle et isocèle
  - pour avancés : un triangle isocèle
  - une maison
- Étant donné que la méthode de dessin manuelle est assez fatigante et ennuyante, ajoutez maintenant les méthode suivantes à la classe **`TurtleBox`** :
  - **`drawSquare(double pSideLength)`**
  - **`drawEquilateralTriangle(double pSideLength)`**
  - **`drawIsocelesRightangledTriangle(double pBase)`**
  - **`drawIsocelesTriangle(double pBase, double pHeight)`** **`//p.avancés`**
  - **`drawHouse(double pBase)`**
- Ajoutez maintenant les méthodes suivantes :
  - **`drawPolygon(double pSideLength, int pN)`** qui dessine un polynôme équilatéral et équiangle de n côtés
  - **`drawCircle(double pRadius)`** qui dessine un cercle de rayon **`pRadius`** tracé de façon aussi précise que possible
  - **`drawCircleCenter(double pRadius)`** qui dessine un cercle de rayon **`pRadius`** tracé de façon aussi précise que possible. Au début et à la fin, la tortue se trouve au milieu du cercle
  - **`drawCar(double pLength)`** qui dessine une voiture

Ajoutez d'autres figures intéressantes qui reposent sur les méthodes déjà présentes.