



Bases de la programmation orientée objet

IUT Montpellier-Sète – Département Informatique

- Cours: M2103 - support [ici](#)
- Enseignants: [Marin Bougeret](#), [Romain Lebreton](#), [Sophie Nabitz](#), [Victor Poupet](#), [Petru Valicov](#)
- Le [forum Piazza](#) de ce cours pour poser vos questions
- [Email](#) pour une question d'ordre privée concernant le cours.
- Le [sujet du TP](#) en format .pdf téléchargeable et imprimable.

TP8

Thème : Modèles objets avancés

Comme d'habitude, cliquez sur le lien ci-dessous pour faire votre fork privé du TP avec GitHub Classroom :

[Lien fork TP8](#)

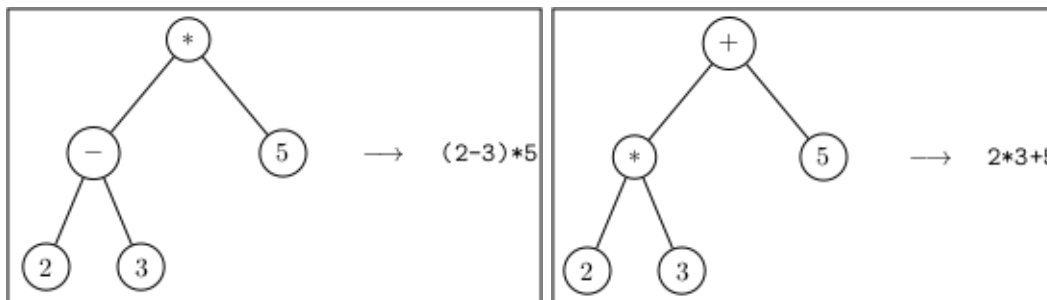
Date limite de rendu de votre code sur le dépôt GitHub : **Dimanche 11 avril à 23h00**

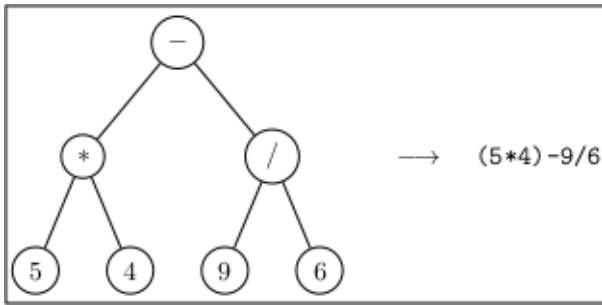
Exercice 1 - Expressions Arithmétiques

Vous allez écrire une petite application qui gère les expressions arithmétiques composées des quatre opérations mathématiques usuelles ($-$, $+$, $/$ et $*$). Pour simplifier on supposera que les opérations de nos expressions arithmétiques sont d'arité 2, c'est-à-dire ont deux opérandes. Par exemple :

- $(2+3)*4-1 \rightarrow$ expression valide
- $5*4-9/6 \rightarrow$ expression valide
- $+(2+3)*4-1 \rightarrow$ expression non valide
- $*1 \rightarrow$ expression non valide

En respectant les priorités des opérateurs et des parenthèses, on peut représenter une expression arithmétique sous forme d'une structure arborescente, comme indiqué ci-dessous (on appelle ces structures *arbres binaires*) :





On souhaite programmer une petite application qui, à partir d'une expression arithmétique donnée sous forme d'arbre binaire, calcule la valeur du résultat de cette expression. Par exemple pour chacun des trois arbres donnés ci-dessus, le résultat devrait être -5, 11 et 18.5 respectivement.

Les premiers bouts de code vous sont donnés dans le package `fr.umontpellier.iut.expressions`. L'interface `Expression` modélise les expressions arithmétiques. Observez la méthode `double calculerValeur()`. Cette méthode doit retourner le résultat du calcul de l'expression arithmétique.

1. Les classes `Operation` et `Nombre` implémentant chacune l'interface `Expression`, vous sont également données. Intuitivement, la classe `Nombre` va correspondre à la feuille de votre arbre binaire, alors que la classe `Operation` va correspondre à un nœud interne de l'arbre. Une `Operation` sera donc composée d'un caractère représentant l'opérateur, et de ses deux sous-expressions.

Ajoutez le code nécessaire à ces deux classes pour modéliser cette information et pour que `double calculerValeur()` retourne le résultat d'évaluation de l'expression correspondant à sa classe. Ainsi, dans la classe cliente (`AppExpression`), on devrait pouvoir construire une expression arithmétique et ensuite calculer son résultat :

```
Expression monExpr;
// pour construire une Expression, il faudra d'abord créer les Nombres,
// puis les Expressions utilisant ces nombres, etc.
double resultat = monExpr.calculerValeur(); // évalue l'expression
System.out.println(resultat);
```

Pensez à écrire suffisamment de tests unitaires pour valider votre solution.

Remarque : Vous supposerez que les seuls caractères possibles pour les opérateurs seront : +, -, * et /

Remarque : Lorsque le calcul implique une division par zéro, vous pouvez vous contenter de lever une `exception` correspondante :

```
throw new ArithmeticException("Division par zéro");
```

2. On souhaite maintenant pouvoir afficher une expression arithmétique en format texte (lisible par l'humain) à partir de l'arbre binaire. Par exemple le texte de l'arbre le plus à gauche devrait être $((2-3)*5)$. En vous inspirant de la solution à la question précédente, redéfinissez la méthode `toString()` dans `Operation` et `Nombre` afin de permettre l'affichage d'une expression arithmétique. À l'utilisation cela devrait ressembler à quelque chose comme ceci :

```
Expression monExpr; // = new... à vous d'écrire le code nécessaire
System.out.println(monExpr); // affiche bien le texte de l'expression
```

Attention : Pensez à respecter les priorités des opérateurs et donc mettre les parenthèses aux bons endroits. Pour vous simplifier la tâche vous pouvez parenthéser toutes les expressions, y compris celles pour lesquelles habituellement on n'ajoute pas les parenthèses, par ex : $(2+3)$ ou $((2*3)+5)$ ou $(8*9)$.

3. Dessinez le diagramme de classes et discutez-en avec votre enseignant. Cette structure est une illustration concrète d'un modèle `Composite`. Le même modèle vous sera utile pour l'exercice suivant, que l'on espère bien plus amusant !

Exercice 2 - Images SVG

On se propose ici d'écrire une application qui permet à un utilisateur de produire des images SVG (extension `.svg`). SVG est une abréviation pour [Scalable Vector Graphics](#), un format XML permettant de décrire de manière textuelle une image. L'image est vue comme un *assemblage de plusieurs composants graphiques* (cercles, rectangles, lignes, etc.), regroupés suivant différents critères.

La philosophie ici est la même que pour une page HTML : la structure est décrite dans un format textuel à l'aide de balises (*tag* en anglais) et c'est l'interpreteur (*renderer* en anglais) qui interprète le code pour "dessiner" le rendu.

Un début de code vous est donné dans le package `fr.umontpellier.iut.svg`. Dans cet exercice les noms des différents éléments du code correspondent aux mots-clés de la norme SVG. Par conséquent, ces noms ne respecteront pas forcément les conventions de nommage du *Java*. C'est également pour cela que l'ensemble des classes, attributs et méthodes sont nommés en anglais. Certaines classes ont déjà du code écrit, d'autres classes seront à compléter. Voici le résumé des 3 classes principales qui vous sont données :

- **Tag** - modélise les balises
- **Style** - regroupe les différents styles graphiques que l'on peut appliquer à une balise (épaisseur du trait, couleur, etc.). Tous les objets de type **Tag** ont un attribut de type **Style**.
- **SVG** - représente la balise `<svg>...</svg>` qui se trouve à la *racine* du fichier SVG (l'équivalent de la balise `<html>...</html>` en HTML).

1. Prenez le temps de regarder et comprendre ce que fait le code fourni pour ces 3 classes. En particulier, essayez de comprendre le fonctionnement de la méthode `toSVG()` de la classe **Tag** et regardez les différents attributs et méthodes des classes **SVG** et **Style**.
2. En SVG les cercles sont représentés par des balises comme dans l'exemple suivant :

```
<circle cx="5" cy="3" r="1.3"/>
```

où `cx`, `cy` et `r` représentent respectivement les coordonnées du centre et le rayon.

La classe **Circle** vous est donnée. La méthode `getParameters()` de **Tag** y est redéfinie.

Attention : la méthode `toSVG()` de la classe **Tag** devra être héritée telle qu'elle et **ne doit pas** être redéfinie.

Générez une image SVG contenant un ou plusieurs cercles. Pour cela, instanciez un objet de type **SVG** dans la classe cliente **AppSVG**. Ajoutez-lui un cercle en utilisant sa méthode `public void add(Tag tag)`. Vous pouvez ensuite générer le fichier SVG avec la méthode `public void saveAsFile(String filename)` de **SVG**. Ce fichier sera généré à la racine de votre projet et vous pouvez le visualiser avec le navigateur ou tout lecteur de fichiers image.

3. De manière similaire complétez la classe **Rectangle**. Voici un exemple de balise rectangle en SVG:

```
<rect x="8" y="6" width="34" height="49"/>
```

Testez dans le programme principal en générant une image SVG correspondante.

4. Ajoutez des *getters* et des *setters* à la classe **Style** afin de pouvoir contrôler les paramètres `stroke`, `fill`, `strokeWidth` et `fillOpacity`. Ajoutez également un *getter* et un *setter* pour l'attribut `style` de la classe **Tag**.
5. Dans la fonction `main(String args[])` de **AppSVG** générez une image contenant un rectangle dont le contour (`stroke`) est bleu et l'intérieur (`fill`) est vert, et un cercle dont le contour est noir et l'intérieur est rouge. En SVG les couleurs peuvent être données directement par leur nom ("`green`", "`black`", etc.) ou au format RGB en hexadécimal ("`#00FF00`" pour vert par exemple).
6. En SVG la balise `<g>...</g>` permet de définir un *groupe*. Les groupes peuvent contenir d'autres balises (qui peuvent être des formes comme cercle, rectangle, etc. mais aussi des groupes). Ceci permet entre autres d'appliquer le style du groupe à tous ses éléments. Complétez la classe **Group** correspondant à cette balise avec :

- un constructeur `public Group()` qui génère un groupe vide ;
- une méthode `public void add(Tag t)` qui permet d'ajouter une balise à la fin du groupe.

Redéfinissez également dans **Group** les méthodes nécessaires héritées de la classe **Tag** afin que sa méthode `toSVG()` fonctionne correctement pour un **Group** (rappel : vous ne devez pas redéfinir la méthode `toSVG()`).

Remarque : Vous pouvez vous inspirer de la classe `SVG` pour écrire `Group`.

- Le problème maintenant est que d'après la spécification SVG, les balises `<g>` et `<svg>`, ne peuvent pas contenir de balise `<svg>`. Proposez une restructuration du code, afin de préserver le comportement programmé précédemment, mais aussi tenir compte de ces nouvelles contraintes. Dessinez le diagramme de classes.
- Si ce n'est pas déjà fait, remarquez que les classes `Group` et `SVG` partagent beaucoup de comportements... Proposez une solution en évitant cette duplication de code.
- Il est possible d'appliquer des *transformations* géométriques (translation, rotation, etc.) aux différentes balises. Les transformations font partie du style (commun à toutes les balises) et s'appliquent l'une après l'autre (cf. [documentation](#)). Exemple d'une balise rectangle à laquelle on applique successivement une mise à l'échelle, une rotation et une transformation :

```
<rect x="0" y="0" width="100.9" height="63.5"
  transform="translate(30 40) rotate(-10 50 100) scale(1.05 0.5)"/>
```

Remarque : En SVG les transformations sont appliquées dans l'ordre inverse du texte contenu dans la balise.

Implémentez les différentes classes de transformations afin que leur méthode `toString()` retourne le texte correspondant à la transformation (par exemple `"translate(5 -12)"` pour une translation). Dans un premier temps vous vous contenterez d'implémenter les classes `Translate`, `Rotate` et `Scale` ayant les constructeurs suivants :

```
public Translate(double x, double y);
public Rotate(double a, double x, double y);
public Scale(double x, double y);
```

- Modifiez la classe `Style` pour qu'elle gère une liste de transformations. Vous devez notamment :
 - implémenter la méthode `addTransform(...)` qui permet d'ajouter au style une transformation passée en paramètre ;
 - modifier la méthode `toSVG()` pour qu'elle affiche correctement l'attribut `transform` (attention à bien respecter l'ordre d'application, la dernière transformation ajoutée avec `addTransform` doit apparaître en premier).
- Les rectangles en SVG peuvent avoir deux attributs optionnels `rx` et `ry` qui permettent d'arrondir les angles du rectangle (cf. [documentation](#)). Modifiez la classe `Rectangle` pour en tenir compte, en ajoutant notamment un constructeur

```
public Rectangle(double x, double y, double width, double height, double rx, double ry);
```

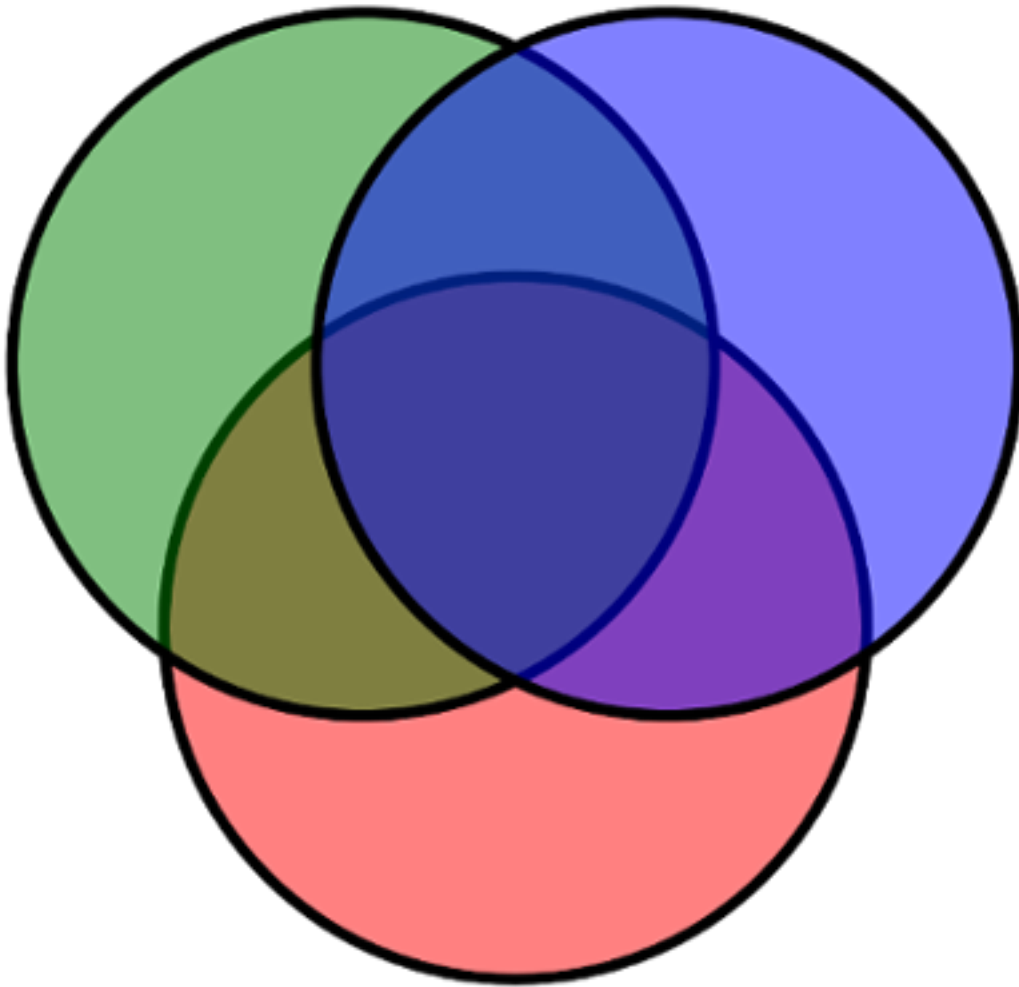
Attention : Ces attributs sont optionnels, il faut donc laisser la possibilité à l'utilisateur de construire un rectangle sans ces nouveaux paramètres.

- À l'aide des classes que vous avez écrites, écrivez un programme qui génère un fichier SVG ayant le contenu suivant :

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="1000.0" height="1000.0">
  <rect x="0.0" y="0.0" width="1000.0" height="1000.0" fill="lightblue"/>
  <g stroke="black" stroke-width="20.0" fill-opacity="0.8">
    <rect x="100.0" y="100.0" width="200.0" height="200.0" fill="#080" />
    <rect x="200.0" y="200.0" width="400.0" height="400.0" fill="#0C0" />
    <g transform="translate(0.0 700.0) rotate(-45 100 50)">
      <rect x="100.0" y="100.0" width="200.0" height="200.0" fill="#800" />
      <rect x="200.0" y="200.0" width="400.0" height="400.0" fill="#C00" />
    </g>
  </g>
</svg>
```

(le code a été indenté pour plus de lisibilité mais vous n'avez pas à générer du code indenté)

13. Écrivez un programme qui génère un fichier SVG représentant l'image ci-dessous :



Remarque : Les trois cercles ont la même épaisseur de trait, couleur de bordure et transparence (50%).

14. Dessinez le diagramme de classes final de votre application.

15. **Bonus :** Écrivez les autres classes de transformation `SkewX`, `SkewY` (cf. [documentation](#)) et d'autres formes géométriques (cf. [Basic shapes](#)).