Domaine Sciences et Technologies
LICENCE INFORMATIQUE : PARCOURS MI

Aix*Marseille université Socialement engagée

Approfondissements en informatique (COO) : Devoir 3
Code UE : SIN6U28T

Année 2023-24

Devoir 3 : Formules

TP noté

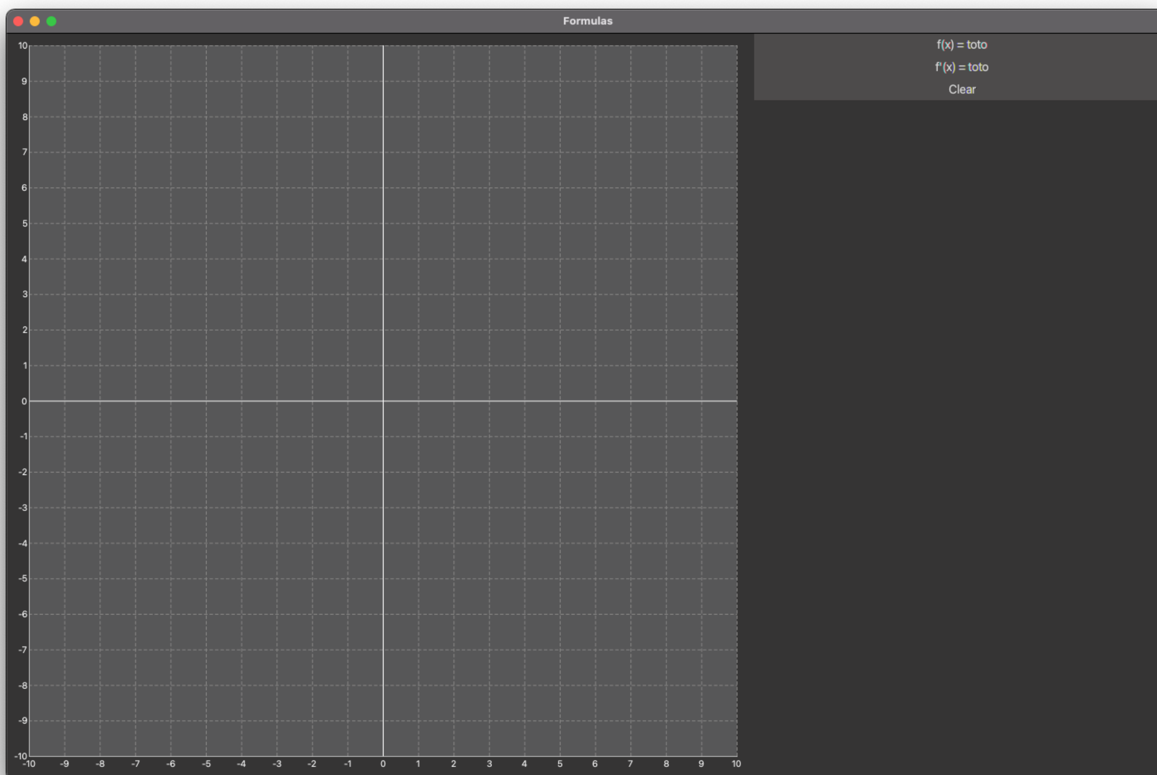
Ce devoir est à faire pour le 1er juin 2024. Il faudra que le dépôt *git* soit à jour pour le 1er juin 2024 à 23h59 et que je (Arnaud Labourel, identifiant `@alaboure`) sois membre du projet avec des droits au moins égal à `reporter`.

Consignes pour démarrer le TP

Comme pour les TP précédents, on va utiliser git pour la gestion de versions et vous allez devoir cloner. Il vous faut donc vous reporter aux consignes du premier TP.

Lien vers le projet gitlab à forker pour le TP : lien projet à forker.

Une fois le dépôt téléchargé, vous pouvez compiler et exécuter le programme en cliquant deux fois sur `formula` -> `application` -> `run`. Vous devriez obtenir l'affichage suivant.



Pour exécuter les tests, il faut passer par l'onglet gradle à droite et cliquer deux fois sur '`color-image` -> `Tasks` -> `verification` -> `test`. Pour le moment, tous les tests ont été désactivés, car certaines classes sont incomplètes.

Objectifs

Dans cette planche de TP, vous allez implémenter des classes pour générer des formules mathématiques. Chaque classe correspondra à un type de formule (constantes, variable, addition, multiplication, . . .).

Chaque classe devra implémenter l'interface Formula suivante :

```java
public interface Formula {
  /**
   * Compute the value of the formula
   *
   * @param xValue the value of the variable x
   * @return the value of the function when the variable x has value {@code
      xValue}
   */
  double eval(double xValue);

  /**
   * Compute a {@code String} representation of the formula.
   * @return the formula as a {@code String}
   */
  String toString();

  /**
   * Compute the derivative of the formula.
   * @return the derivative of the formula
   */
  Formula derivative();
}
```

Une classe implémentant `Formula` devra donc avoir trois fonctionnalités : - le calcul de sa valeur étant donné une valeur pour la variable x : méthode `eval`, - la représentation en chaîne de caractères de la formule : méthode `toString`, - le calcul de sa dérivée sous la forme d'une autre formule : méthode `derivative`.

Constante

Le contrat

Vous allez commencer par corriger la classe `Constant` représentant une constante. Cette classe permet de construire une formule correspondant à une constante. Cette classe implémente l'interface `Formula` et contient :

— un constructeur `public Constant(double value)` permettant de créer une constante avec une certaine valeur,

— une méthode `public double eval(double xValue)` qui devra toujours retourner la valeur de la constante,

— une méthode `public String toString()` qui devra retourner la chaîne de caractères correspondant à la constante,

— une méthode `public Formula derivative()` qui devra retourner une formule qui est la dérivée de la constante (indice, pour n'importe quelle fonction $f(x) = c$ avec c constante, la dérivée de la fonction f est la fonction constante définie par $f'(x) = 0$).

Tâche 1 : Corriger la classe `Constant` de sorte qu'elle passe les tests présents dans la classe `ConstantTest` dans `src/java/main/formula` (il vous faudra réactiver les tests en enlevant les `@Disabled`).

Affichage

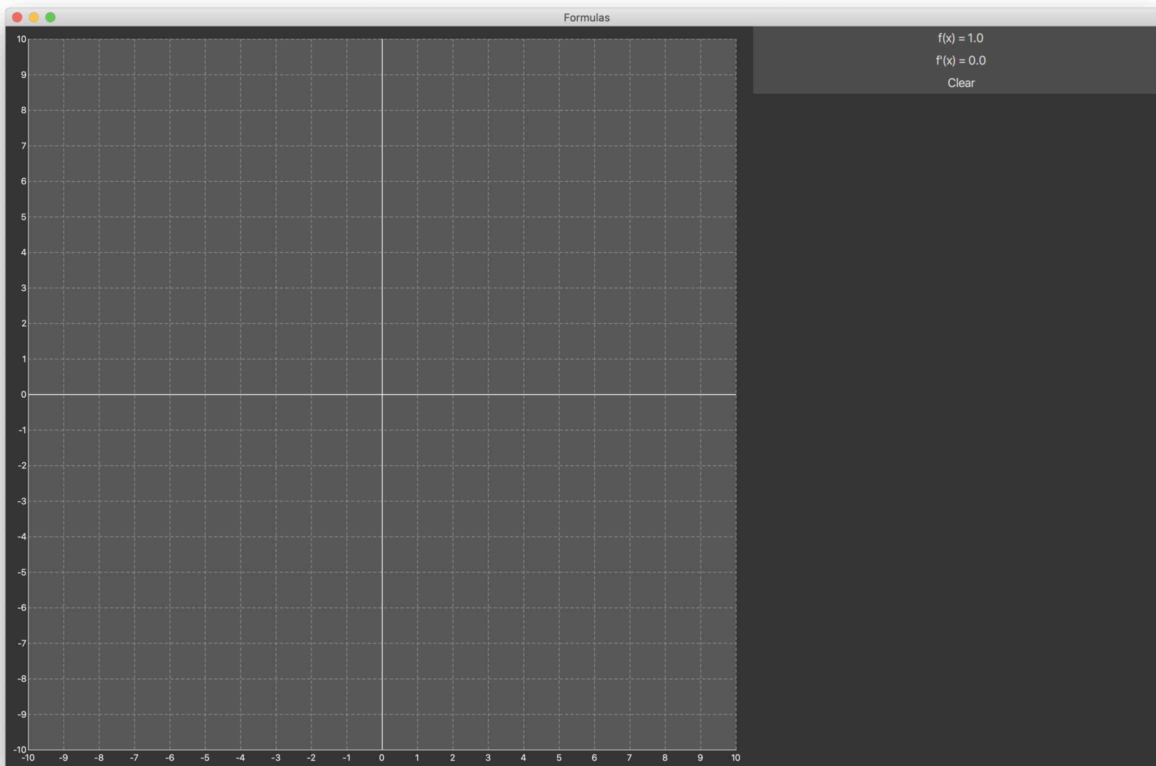
Pour tester votre classe constante, il faut exécuter la tâche `run`. Les fonctions affichées sont définies par les deux lignes suivantes dans le constructeur de la classe `viewer.FunctionList` :

```
1 PlottableFunction function = new PlottableFunction(new Constant(1), "f");
2 addFunctionAndItsDerivative(function);
```

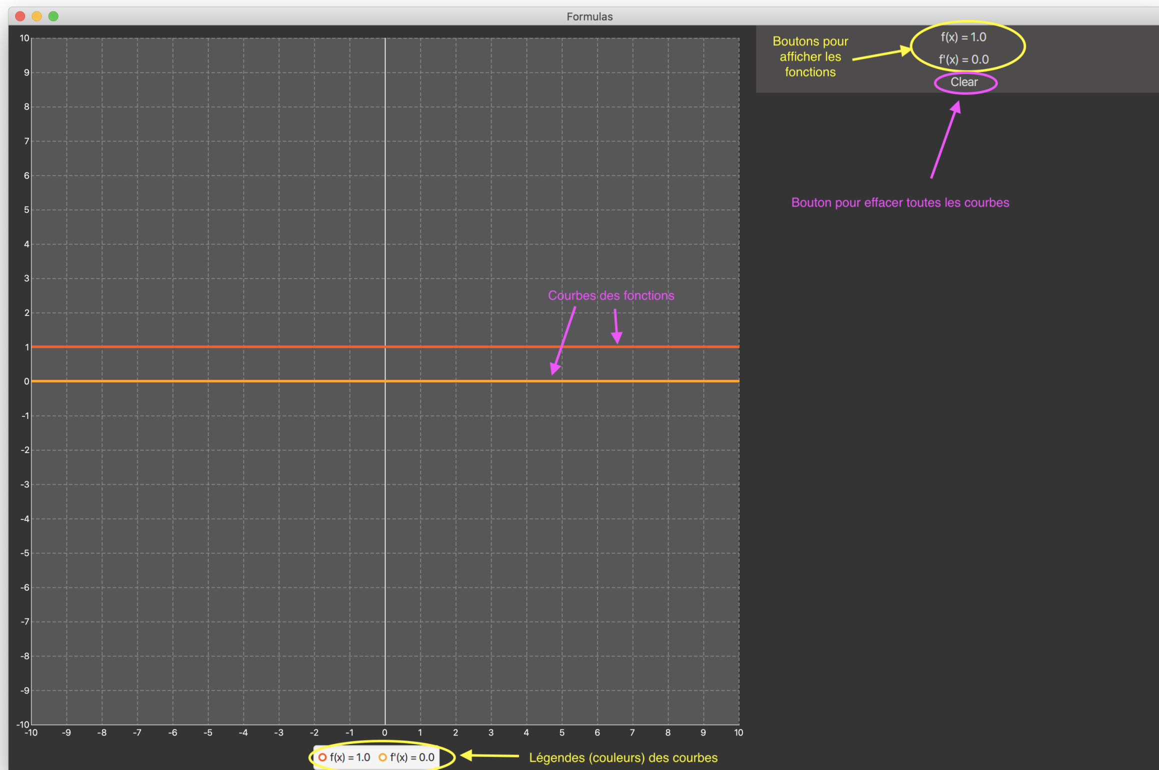
Ces deux lignes permettent respectivement :

— de créer une fonction f dont la formule est une constante égale à 1 et ayant comme nom f,
— puis de rajouter la fonction f et sa dérivée f'.

Vous devriez obtenir l'affichage suivant.



Pour afficher les fonctions, il suffit de cliquer sur les boutons correspondants. Vous devriez obtenir l'affichage suivant :



Tâche 2 : Rajoutez les lignes indiquées ci-dessus et vérifiez que l'affichage obtenu est correct.

Variable x

Le contrat

Vous allez maintenant définir une nouvelle classe `VariableX` représentant une variable x. Cette classe permettra de construire une formule correspondant à la variable x.

Cette classe implémentera l'interface `Formula` et devra contenir :

— un constructeur `public VariableX()` permettant de créer une variable,

— une méthode `public double eval(double xValue)` qui devra toujours retourner la valeur `xValue`,

— une méthode `public String toString()` qui devra retourner la chaîne de caractères correspondant à la variable x et donc la chaîne de caractère `"x"`,

— une méthode `public Formula derivative()` qui devra retourner une formule qui est la dérivée en x

de la variable x (indice : pour une fonction $g(x) = x$, la dérivée de la fonction g est $g'(x) = 1$).

Tâche 3 : Ajouter la classe `VariableX` dans `src/java/main/formula`.

Le test

Vous pouvez tester votre classe en créant une classe `VariableXTest` s'inspirant de la classe `ConstantTest`.

Tâche 4 : Ajouter une classe `VariableXTest` dans `src/java/test/formula` qui devra tester le bon comportement de la classe `VariableX`.

Affichage

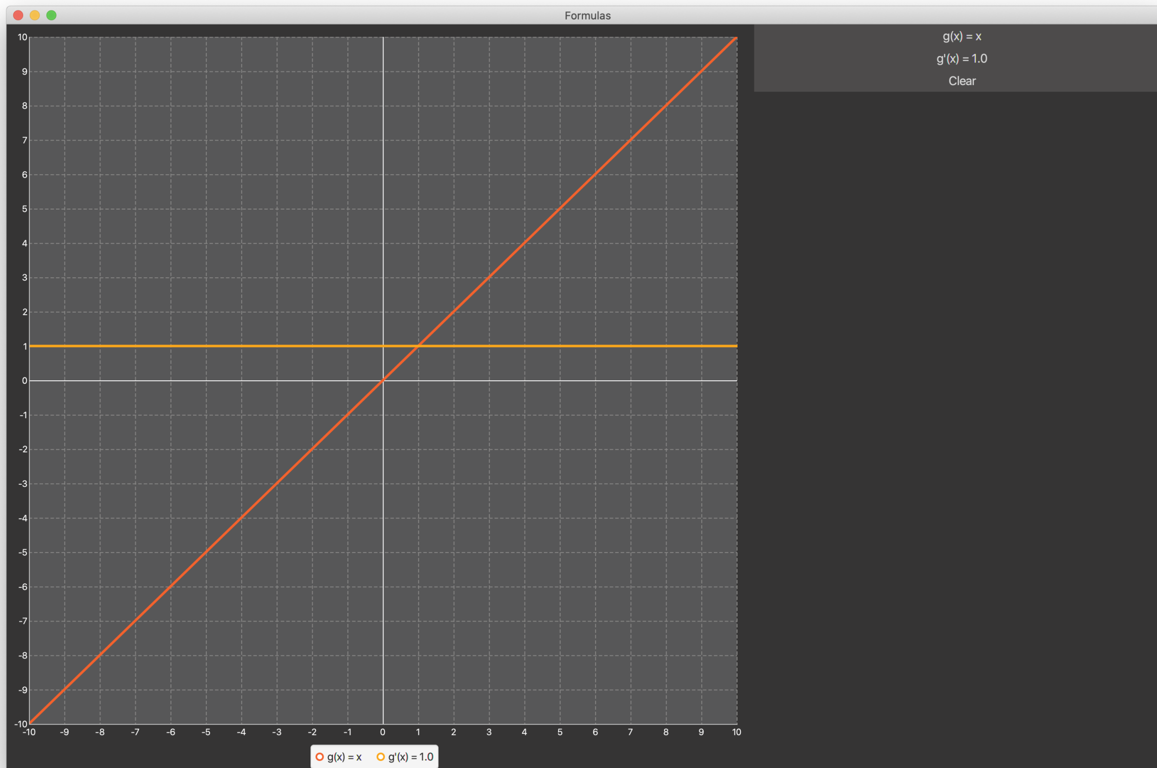
Pour tester votre classe variable, vous aller créer une fonction égale à x dans le logiciel de dessin de fonctions. Pour cela, il vous faut modifier le constructeur de la classe `viewer.FunctionList` pour rajouter les deux lignes suivantes à la place du `//TODO` :

```
1 PlottableFunction function = new PlottableFunction(new VariableX(), "g");
2 addFunctionAndItsDerivative(function);
```

Ces deux lignes permettent respectivement :

— de créer une fonction dont la formule est égale à x et ayant comme nom g,
— puis de rajouter la fonction g et sa dérivée g'.

Vous devriez obtenir l'affichage suivant.



Tâche 5 : Rajoutez les lignes indiquées ci-dessus et vérifiez que l'affichage obtenu est correct.

Addition

Le contrat

Vous allez maintenant définir une classe `Addition` représentant une addition de deux formules. Cette classe permettra de construire une formule correspondant à la somme de 2 formules.

Cette classe implémentera l'interface `Formula` et devra contenir :

— un constructeur `public Addition(Formula leftMember, Formula rightMember)` permettant de créer l'addition des deux formules données en arguments correspondants aux deux membres sommés,

— une méthode `public double eval(double xValue)` qui devra toujours retourner la somme des valeurs des deux membres de l'addition,

— une méthode `public String toString()` qui devra retourner la chaîne de caractères correspondant à l'addition, soit la chaîne de caractères correspondant au membre de gauche, concaténée avec le symbole `+` puis concaténé au membre de droite,

— une méthode `public Formula derivative()` qui devra retourner une formule qui est la dérivée en x de la formule (indice : pour une fonction $h(x) = f(x)+g(x)$, la dérivée de la fonction h est $h'(x) = f'(x)+g'(x)$).

Tâche 6 : Ajouter la classe `Addition` dans `src/java/main/formula`.

Le test

Tâche 7 : Ajouter une classe `AdditionTest` dans `src/java/test/formula` qui devra tester le bon comportement de la classe `Addition`.

Affichage

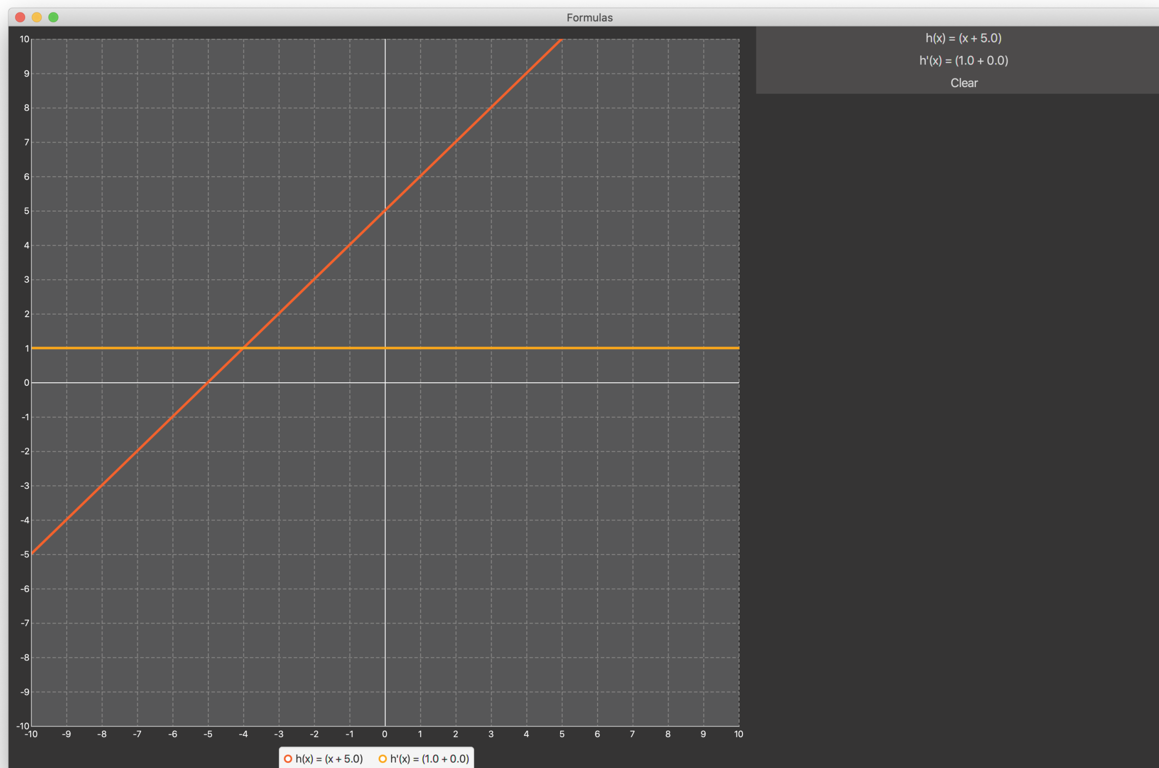
Pour tester votre classe addition, vous aller créer une fonction égale à $x+5$ dans le logiciel de dessin de fonctions. Pour cela, il vous faut modifier le constructeur de la classe `viewer.FunctionList` pour rajouter les deux lignes suivantes à la place du `//TODO` :

```
1 PlottableFunction function =
2   new PlottableFunction(new Addition(new VariableX(),
3                              new Constant(5)), "h");
4 addFunctionAndItsDerivative(function);
```

Ces deux lignes permettent respectivement :

— de créer une fonction dont la formule est égale à $x+5$ et ayant comme nom h,
— puis de rajouter la fonction h et sa dérivée h'.

Vous devriez obtenir l'affichage suivant.



Tâche 8 : Rajoutez les lignes indiquées ci-dessus et vérifiez que l'affichage obtenu est correct.

Multiplication

Vous allez maintenant définir une classe `Multiplication` représentant une multiplication de deux formules. Cette classe permettra de construire une formule correspondant au produit de 2 formules.

Cette classe implémentera l'interface `Formula` et devra contenir :

— un constructeur `public Multiplication(Formula leftMember, Formula rightMember)` permettant de créer la multiplication des deux formules données en arguments correspondants aux deux membres multipliés,

— une méthode `public double eval(double xValue)` qui devra toujours retourner le produit des valeurs des deux membres de la multiplication,

— une méthode `public String toString()` qui devra retourner la chaîne de caractères correspondant à la multiplication, soit la chaîne de caractères correspondant au membre de gauche, concaténée avec le symbole `*` puis concaténé au membre de droite,

— une méthode `public Formula derivative()` qui devra retourner une formule qui est la dérivée en x de la formule (indice : pour une fonction $h(x) = f(x)g(x)$, la dérivée de la fonction h est $h'(x) = f(x)g'(x) + f'(x)g(x)$).

Tâche 9 : Ajouter la classe `Multiplication` dans `src/java/main/formula`.

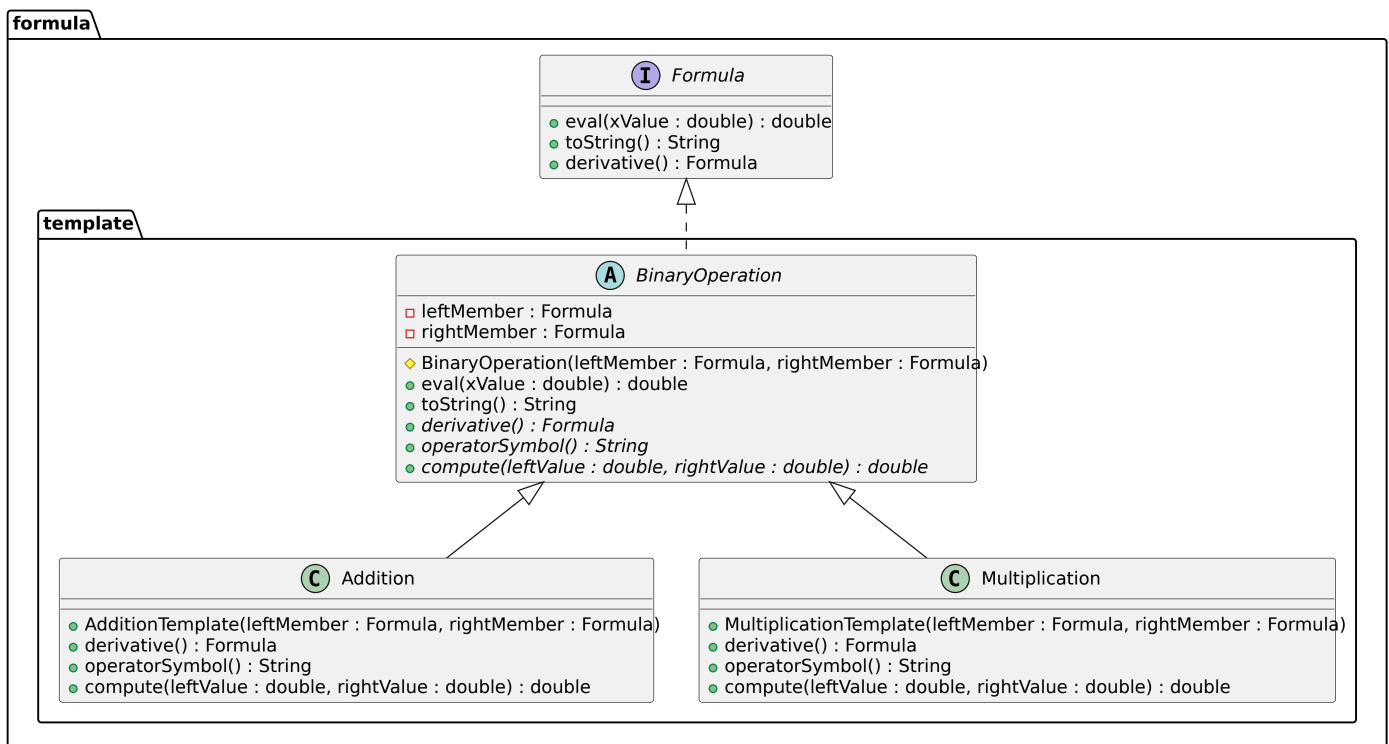
Le test

Tâche 10 : Ajouter une classe `MultiplicationTest` dans `src/java/test/formula` qui devra tester le bon comportement de la classe `Multiplication`.

Utilisation de patron de méthode

Les classes `Addition` et `Multiplication` que vous avez écrites ont beaucoup en commun. Comme nous l'avons vu dans le cours, la répétition est quelque chose qu'un bon programmeur essaye d'éviter. Afin de remédier à cela, on peut utiliser le patron de méthode. Patron de Méthode est un patron de conception comportemental qui permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.

L'objectif ici est donc de créer une classe abstraite `BinaryOperation` et des nouvelles implémentations des classes pour l'addition et la multiplication appelée `Addition` et `Multiplication`. Toutes ces classes devront être dans un nouveau package `template` sous-package de `formula` (répertoire `src/java/main/formula/template`).



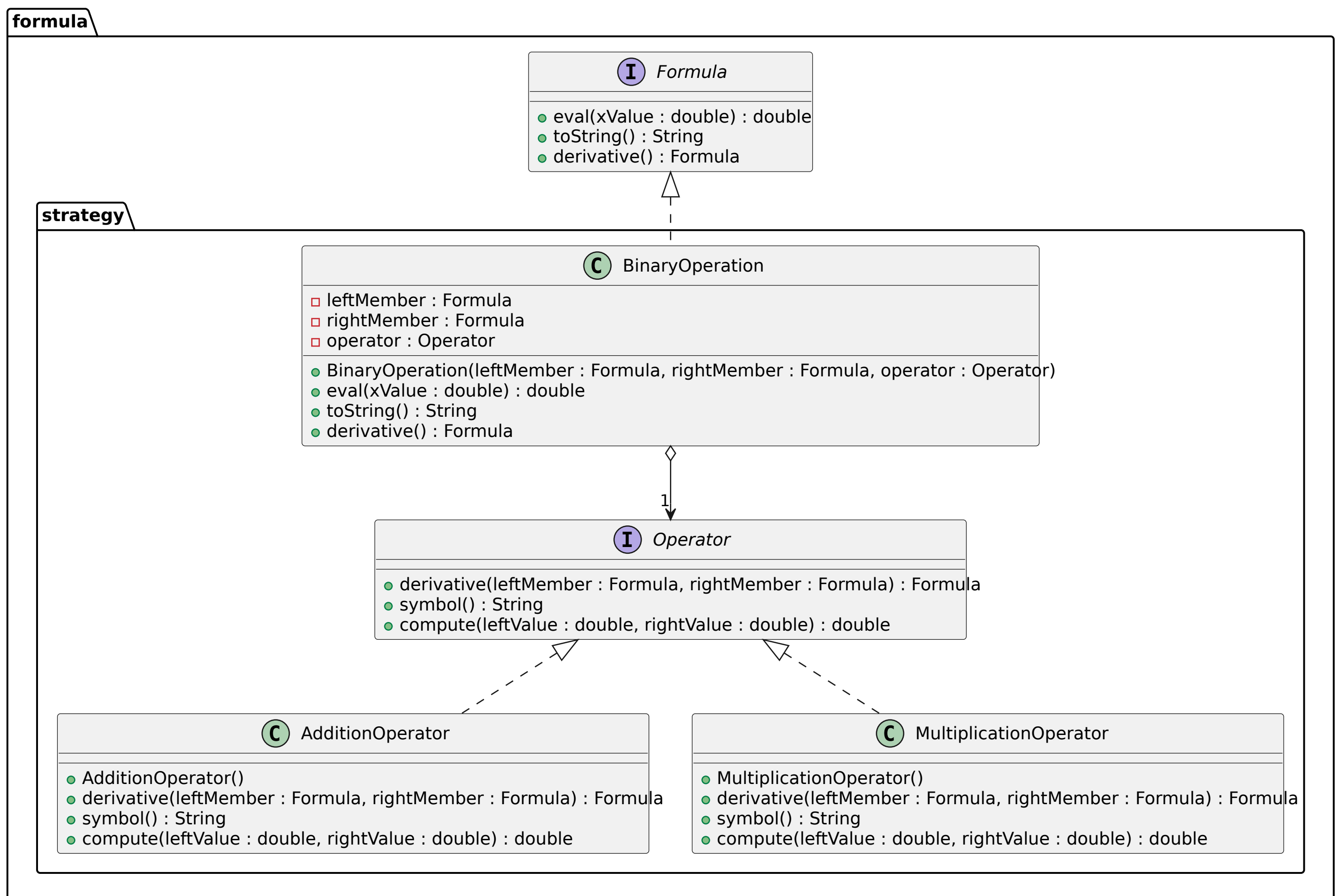
La méthode `operatorSymbol` renvoie le symbole représentant l'opérateur de l'opération (`"+"` ou `"*"`) et la méthode `compute` calcule le résultat de l'opération sur deux valeurs.

Tâche 11 : Ajoutez les classes `BinaryOperation`, `Addition` et `Multiplication` dans `src/java/main/formula/template`.

Utilisation de stratégie

Un autre patron de méthode que l'on peut utiliser pour éviter la duplication de code est le patron de conception stratégie. Stratégie est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

L'objectif ici est donc de créer une classe `BinaryOperation`, une interface `Operator` qui contiendra les méthodes spécifiques à l'addition et la multiplication, et des classes implémentant cette interface appelée `AdditionOperator` et `MultiplicationOperator`. Toutes ces classes devront être dans un nouveau package `strategy` sous-package de `formula` (répertoire `src/java/main/formula/strategy`).



Les méthodes d'`Operator` sont les suivantes :

— `symbol` renvoie le symbole représentant l'opérateur (`"+"` ou `"*"`) ;

— `derivative` renvoie la formule dérivée de l'opérateur appliqué sur les deux formules `leftMember` et `rightMember` ;

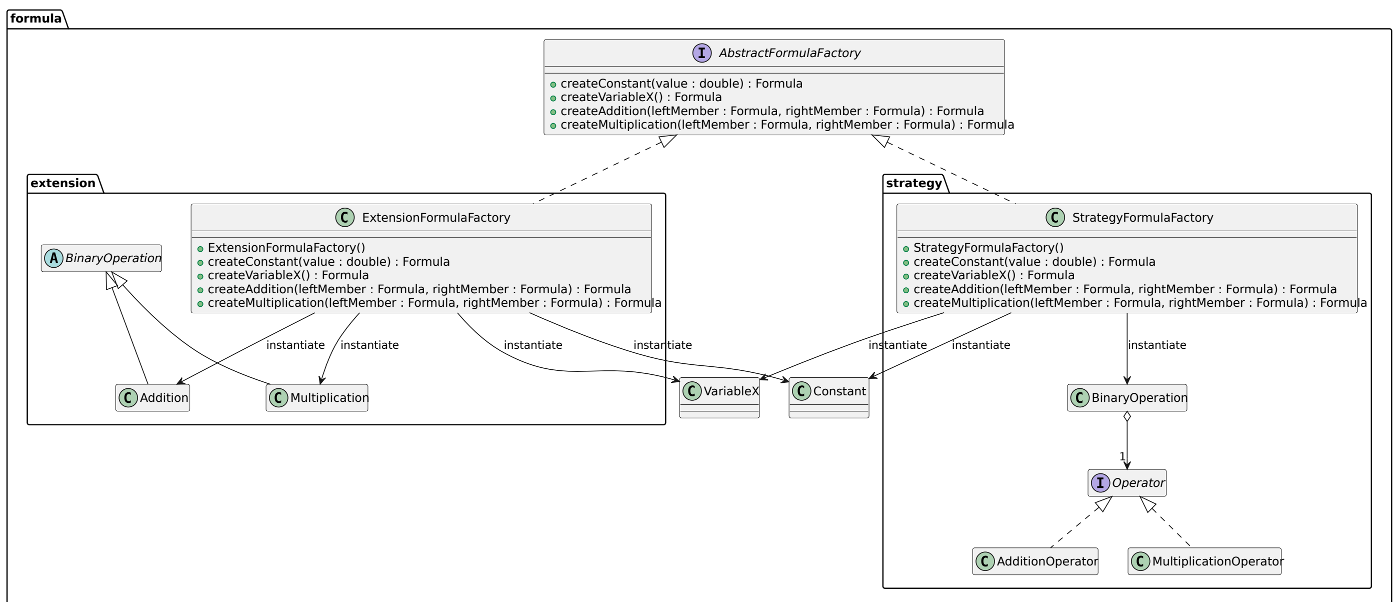
— `compute` calcule le résultat de l'opérateur appliqué sur les deux valeurs.

L'objectif est d'avoir une formule équivalente à une instance d'`Addition` avec des membres `leftMember` et `rightMember` en créant une `BinaryOperation` avec `new BinaryOperation(leftMember, rightMember, new AdditionOperator())`.

Tâche 12 : Ajoutez l'interface `Operator` et les classes `BinaryOperation`, `AdditionOperator` et `MultiplicationOperator` dans `src/java/main/formula/strategy`.

Fabrique abstraite

Maintenant qu'on a deux versions possibles pour les multiplications et les additions, il nous faudrait un code qui nous permette de changer facilement d'une version à une autre. Pour cela, on peut utiliser fabrique abstraite qui est un patron de conception permettant de créer des familles d'objets apparentés sans préciser leur classe concrète. On va donc définir une interface `AbstractFormulaFactory` qui contiendra quatre méthodes permettant de créer des formules correspondant à des constantes, des variables, des additions et des multiplications. Cette interface aura deux implémentations `ExtensionFormulaFactory` et `StrategyFormulaFactory`. Cela nous donne l'architecture décrite par le diagramme de classe ci-dessous :



Tâche 13 : Ajoutez l'interface `AbstractFormulaFactory` et les classes `ExtensionFormulaFactory` et `StrategyFormulaFactory`.

Tâche 14 : Ajoutez un attribut `formulaFactory` de type `AbstractFormulaFactory` dans `FunctionChart`. Instanciez cet attribut dans le constructeur de `FunctionChart` avec une des

deux classes possibles. Remplacez dans `FunctionChart` les appels aux constructeurs des classes implémentant `Formula` par des appels aux méthodes d'`AbstractFormulaFactory`.