Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Programmation et TP

T1A

Réseau et Sécurité

S19 interfaces, classes abstraites et internes

Roten Marc

2017/2018

Table des matières

0 Introduction

Avec ce travail, on va s'occuper des interfaces, decouvrir en pratique ce que sont les classe abstraites et internes.

1 Exercice 1 PlayWithSeries

Au lieu d'une méthode très spécifique, on va recourir à une classe abstraite pour séparer ce qu'il y a de commun à toute "série mathématique" (énumérer des valeurs de *i* et faire des additions) et ce qui est propre à une formule particulière.

```
public abstract class Series {
  public abstract double function(int i);
  public double evaluateBetween(int from, int to) {…}
}
```

1ª Compléter la classe abstraite `Series`.

1ᵇ Ecrire un programme `PlayWithSeries` contenant deux sous-classes internes concrètes `SeriesB1` et `SeriesB2` dédiées respectivement aux deux formules ci-contre. Le programme évaluera ces séries entre 0 et 20.

`Math.pow()`

$$\sum \frac{i}{2^i} \qquad \sum \frac{i^2}{2^i}$$

classe playWithSeries

```
public class PlayWithSeries {
  public static abstract class Series {
    public abstract double function(int i);
    public double evaluateBetween(int from, int to) {
      double sum=0;
      for(int i=from; i<=to; i++) {
        sum += function(i);
      }
      return sum;
    }
  }
  //================================================
   static class SeriesB1 extends Series{
   @Override
   public double function(int i) {

     return 1/Math.pow(2, i);
   }
   }
   static class SeriesB2 extends Series{
   @Override
   public double function(int i) {
     return (Math.pow(i,2))/(Math.pow(2, i));
   }
   }
  //================================================
  public static void main(String[] args) {
    // ****************************************
```

```
    // ****** With internal static classes: ******
    // ********************************************
     Series sa=new SeriesB1();
     Series sb=new SeriesB2();
     System.out.println(sa.evaluateBetween(0, 20));
     System.out.println(sb.evaluateBetween(0, 20));
     System.out.println();
  }
}
```

2 Ex2 PlayWithFunctions

A part la somme, on peut appliquer d'autres opérations "à la chaîne" sur une suite de valeurs. On part d'une valeur initiale (qui rappelle la notion mathématique d'*élément neutre*), puis on cumule successivement le résultat courant avec la prochaine valeur. L'interface `IFoldableOperation` est une manière possible de spécifier ce mécanisme (ça évoque vaguement une feuille pliée en accordéon).
Voir illustration au verso.

```
public interface IFoldableOperation {
  double initialValue();
  double combine(double accumulated,
                 double newValue);
}
```

```
    sum: ((0 + v1) + v2) + v3
product: ((1 * v1) * v2) * v3
```

2ª Implémenter la classe `FoldOperators` dont la spécification (membres `public`) se limite à trois constantes. Cette classe contiendra 3 classes internes pour définir chacune des 3 façons spécifiques de faire émerger un résultat : sum/product/max.

Dans une série mathématique, les termes à combiner dépendent de valeurs (de *i*) entières et consécutives, mais ça peut être considéré comme un cas particulier. L'interface `INumberSequence` généralise l'idée d'une succession de nombres.

```
public class FoldOperators {
  public static final IFoldableOperation SUM
                              = new SumOperation();
  public static final IFoldableOperation PRODUCT=…;
  public static final IFoldableOperation MAX=…;

  private static class SumOperation
            implements IFoldableOperation {…}
}
```

```
public interface INumberSequence {
  boolean hasMoreNumbers();
  double  nextNumber();
}
```

classe FoldOperators

```
public class FoldOperators {
  // ************************************
  // ****** With internal classes ******
  // ************************************

  public static final IFoldableOperation SUM = new
     SumOperation();
  public static final IFoldableOperation PRODUCT = new
     ProductOperation();
  public static final IFoldableOperation MAX = new
     MaxOperation();

  private static class SumOperation implements
     IFoldableOperation{
   @Override
```

```
  public double initialValue() {
    return 0.0;
  }
  @Override
  public double combine(double accumulated, double newValue) {
    return accumulated+newValue;
  }
 }
 private static class ProductOperation implements
    IFoldableOperation{
  @Override
  public double initialValue() {
    return 1.0;
  }
  @Override
  public double combine(double accumulated, double newValue) {
    return accumulated*newValue;
  }
 }
 private static class MaxOperation implements
    IFoldableOperation{
  @Override
  public double initialValue() {
    return Double.NEGATIVE_INFINITY;
  }
  @Override
  public double combine(double accumulated, double newValue) {
    return Math.max(accumulated, newValue);
  }
 }
}
```

2b Implémenter dans `PlayWithFunction` une méthode générale capable de "plier" une succession de nombres en appliquant systématiquement un certain calcul.

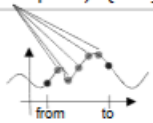
```
public static double foldLeft(INumberSequence ns,
                              IFoldableOperation op) {…}
```

L'interface `IFunction` généralise la notion de fonction réelle à un argument.

```
public interface IFunction {
  double valueAt(double x);
}
```

2c Implémenter la classe suivante qui fournit deux sortes différentes de séquences de nombres. Cette classe contiendra 2 classes internes.
Facultatif ajouter une troisième sorte qui évalue une certaine fonction réelle en différents points.

```
public class CommonSequences {
 public static INumberSequence fromArray(double[] t) { return new ArrayNumberSequence(t); }
 public static INumberSequence fromSeries(IFunction f, int from, int to) { return new … }
 public static INumberSequence fromSampledFunction(IFunction f, double from, double to,
                                                   int nSubSamples) { … }
```



2d Ecrire dans `PlayWithFunction` un main() qui utilise ce qui précède pour calculer :

- **d1** le produit des nombres {1.2, 3.4, 5.6};
- **d2** la série ci-contre (point de départ de la série d'exercices); $\sum_{i=0}^{20} \frac{i}{2^i}$
- **d3** *Facultatif* une estimation (prendre 1002 échantillons) du maximum de la fonction $f(x)=\sin^2(x)*\cos(x)$ entre 0 et π.

deux classes internes pour les fonctions

```
Math.sin()
Math.cos()
Math.PI
```

classe PlayWithFunction

```
package s19;

public class PlayWithFunctions {
  public interface IFoldableOperation {
    double initialValue();
    double combine(double accumulated, double newValue);
  }
  //================================================
  public interface INumberSequence {
    boolean hasMoreNumbers();
    double nextNumber();
  }
  //================================================
  public interface IFunction {
    double valueAt(double x);
  }
  //================================================
  static class MySeriesFct implements IFunction{
    @Override
    public double valueAt(double x) {
      return x/Math.pow(2, x);
    }
  }
  static class MyTrigonometricFct implements IFunction{

    @Override
    public double valueAt(double x) {
      return Math.pow(Math.sin(x), 2)*Math.cos(x);
    }

  }
  //================================================
  public static double foldLeft(INumberSequence ns,
     s19.IFoldableOperation product) {
    double result = product.initialValue();

    while(ns.hasMoreNumbers()) {
      result = product.combine(result, ns.nextNumber());
    }

    return result;
  }
  public static void main(String[] args) {
    IFunction fa = new MySeriesFct(), fb = new
       MyTrigonometricFct();
```

```
    double[] doubleArray = {1.2, 3.4, 5.6};
    INumberSequence nsArray =
       CommonSequences.fromArray(doubleArray);
    INumberSequence nsSeries = CommonSequences.fromSeries(fa,
       0, 20);
    INumberSequence nsSamples =
       CommonSequences.fromSampledFunction(fb, 0, Math.PI,
       1000);

    System.out.println(foldLeft(nsArray,
       FoldOperators.PRODUCT));
    System.out.println(foldLeft(nsSeries, FoldOperators.SUM));
    System.out.println(foldLeft(nsSamples, FoldOperators.MAX));
  }
}
```

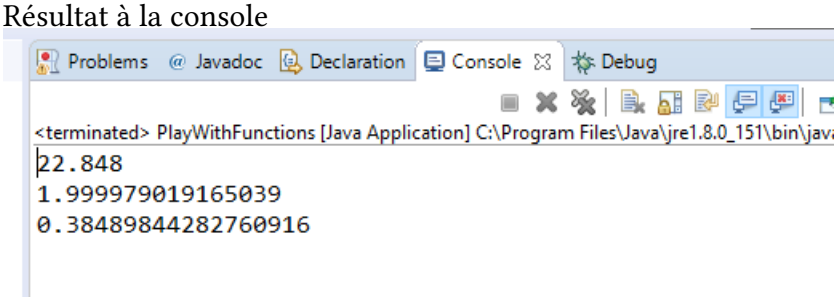
2.1 classe commonSequences

```
public class CommonSequences {
  // ************************************
  // ****** With internal classes ******
  // ************************************

  public static INumberSequence fromArray(double[] t) {
    return new ArrayNumberSequence(t);
  }

  public static INumberSequence fromSeries(IFunction f, int
     from, int to) {
    return new SeriesNumberSequence(f,from,to);
  }

  public static INumberSequence fromSampledFunction(IFunction f,
      double from, double to,
      int nSubSamples) {
    return new
       SampledFunctionNumberSequence(f,from,to,nSubSamples);
  }
  static class ArrayNumberSequence implements INumberSequence{
    private int x = 0;
    private final double[] ANSPrivate;
    public ArrayNumberSequence(double[] t) {
      this.ANSPrivate = t;
    }
    @Override
    public boolean hasMoreNumbers() {
```

```
    if(x<ANSPrivate.length) {
      return true;
    }else return false;
  }
  @Override
  public double nextNumber() {
    return ANSPrivate[x++];
  }

}
static class SeriesNumberSequence implements INumberSequence{
  private final IFunction myIFunctPrivate;
  private int fromPrivate;
  private final int toPrivate;
  public SeriesNumberSequence(IFunction f, int from, int to) {
    this.myIFunctPrivate=f;
    this.fromPrivate = from;
    this.toPrivate = to;
  }
  @Override
  public boolean hasMoreNumbers() {
    if(fromPrivate<=toPrivate) {
      return true;
    }else return false;
  }
  @Override
  public double nextNumber() {
    return myIFunctPrivate.valueAt(fromPrivate++);
  }
}
 static class SampledFunctionNumberSequence implements
    INumberSequence {
     private final IFunction myPrivateFunc;
     private final double privateFrom, privateStep;
     private final int nSamplesTotal;
     private int x;

     public SampledFunctionNumberSequence(IFunction f, double
        from, double to, int nSubSamples) {
       this.myPrivateFunc = f;
       this.privateFrom = from;
       this.nSamplesTotal = nSubSamples + 2;
       this.privateStep = (to - from) / (nSubSamples + 1);
       this.x = 0;
     }

     @Override
     public boolean hasMoreNumbers() {
       return x < nSamplesTotal;
```

```
        }

        @Override
        public double nextNumber() {
          return myPrivateFunc.valueAt(privateFrom + privateStep
              * x++);
        }
      }
}
```

Résultat à la console



3 Conclusion

Grâce à ce Travail, on a pu mieux comprendre et utiliser les interfaces, les interfaces contenant des methodes abstraites.