MOOC Intro POO Java
Corriges semaine 7

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 23 : Encore des employés

La difficulté dans cet exercice était principalement liée au fait que l'on souhaite pouvoir ajouter des nouvelles classes à notre hiérachie d'employés tout en laissant invariant le code déjà développé pour muter un employé d'une classe à l'autre. L'interêt évident d'une telle démarche est que le code réalisé pour la mutation est générique et n'a donc pas besoin de faire l'objet de modification/adjonction, quelque soit l'évolution de notre hiérarchie d'employés. La solution simple qui consiste à coder explicitement une méthode spécifique pour chaque type de mutation possible est donc écartée.

Pour muter un employé d'une catégorie à une autre il va falloir récupérer toutes les informations nécessaires pour mettre à jour ses nouveaux attributs. Certains de ces attributs sont communs et ne posent donc pas de problèmes. Pour les attributs spécifiques aux sous-classes par contre, il va falloir les mettre à jour à partir des données de l'ancienne catégorie. Dans la solution qui vous est proposée ci-dessous, chaque classe d'employés fournit un constructeur prenant en paramètre un objet de type `Employe`. Cet objet contiendra les informations liées à une ancienne catégorie possible de l'employé, et seront utilisées pour mettre à jour les attributs spécifiques. Ainsi pour muter un employé, on construira un employé de la nouvelle catégorie à partir de l'objet de l'ancienne catégorie (en utilisant justement le constructeur précité). La construction d'un nouvel objet à partir d'un ancien utilise le polymorphisme pour permettre un codage générique du calcul des valeurs des nouveaux attributs.

Il existe plusieurs autres façons de procéder.
Par exemple, il serait aussi possible de n'utiliser que le constructeur de base du type du nouvel employé, et ensuite d'appeler une méthode `transfererSalaire` permettant de mettre à jour les informations liées au changement de classe.

Autre variante possible, et qui correspond à une façon typique de faire en orienté-objet lorsque la catégorie d'un objet peut changer au cours de son existence, est d'utiliser l'encapsulation : une classe pour les employés et une hiérarchie de classes pour les catégories (comprenant alors les méthodes de calcul de salaire et de transfert). Un employé aurait donc un attribut lié à la catégorie.

Libre à vous d'experimenter ces variantes.

Voici donc un des codages possibles de la solution à cet exercice. Elle correspond à la première démarche décrite ci-dessus.

Le code se trouve ci-dessous:

```
//Exemple d'utilisation.
class EmployeMain {

  public static void main(String[] args) {
    Gestion gestion = new Gestion();

    // On cr'ee un employ'e Temporaire "T1", et on l'embauche
    Temporaire t1 = new Temporaire("T1", 21, 20);
    gestion.embaucher(t1);
    gestion.afficherEmploye("T1");

    // On le mute dans la categorie Permanent
    gestion.muter(new Permanent(t1, 0, false, 110, 4200));
    gestion.afficherEmploye("T1");

    System.out.println();

    // On cr'ee un employ'e temporaire "P3", et on l'embauche
    Permanent p3 = new Permanent("P3", 1, true, 100, 2000, 20);
    gestion.embaucher(p3);
    gestion.afficherEmploye("P3");

    // On le mute dans la categoire de Temporaire
    gestion.muter(new Temporaire(p3, 20));
    gestion.afficherEmploye("P3");
  }
}


/**
```

```java
 * Contient la liste des employ'es, et les m'ethodes pour les g'erer,
 * avec les m'ethodes embaucher, licencier et afficherEmployer
 * Pour muter un employ'e, il faut cr'eer le nouvel employ'e en utilisant
 * le constructeur a` ce effet, et le notifier a` Gestion en appelant la
 * m'ethode muter.
 * Par exemple:
 * Temporaire e = new Temporaire("T1", 20, 22);
 * gestion.embaucher(e);
 * Permanent eMute = new Permanent(e,0,false,1000,1200);
 * gestion.muter(eMute);
 */
class Gestion {
  private Employe [] employes = new Employe [100];
  final static int nombreJoursMois = 20;

  /**
   * Ajoute l'employ'e dans <employes>
   */
  public void embaucher(Employe employe) {
    // On ne peut avoir qu'un seul employ'e avec le meme nom
    if (getIndexOfEmploye(employe.getNom()) != -1) {
      System.out.println("Il y a d'eja` un employ'e nomm'e " + employe.getNom()
             + " dans gestion");
      return;
    }

    // Trouver le premier emplacement vide (i.e. null) dans le tableau
    for (int i = 0; i < employes.length; i++) {
      if (employes [i] == null) {
          employes [i] = employe;
          return;
      }
    }

    System.out.println("Votre 'equipe est au complet");
  }

  /**
   * Cette m'ethode utilitaire retourne l'index d'un employ'e dans le tableau
   * @param nom nom de l'employ'e a` trouver
   * @return l'index de l'employ'e, ou -1 s'il n'est pas dans la liste
   */
  private int getIndexOfEmploye(String nom) {
    for (int i = 0; i < employes.length; i++) {
      if ((employes [i] != null) && (employes [i].getNom().equals(nom))) {
          return i;
      }
    }
    return -1;
  }

  /**
   * Licencie l'employ'e de Gestion
   */
  public void licencier(Employe employe) {
    int index = getIndexOfEmploye(employe.getNom());
    if (index == -1) {
      System.out.println("Il n'y a pas d'employ'e nomm'e " + employe.getNom());
      return;
    }

    employes[index] = null;
  }

  /**
   * Retourne l'employ'e connaissant son nom
   */
  public Employe getEmploye(String nom) {
    int index = getIndexOfEmploye(nom);
    if (index == -1) {
      System.out.println("Il n'y a pas d'employ'e nomm'e " + nom);
      return null;
    }

    return employes[index];
  }
```

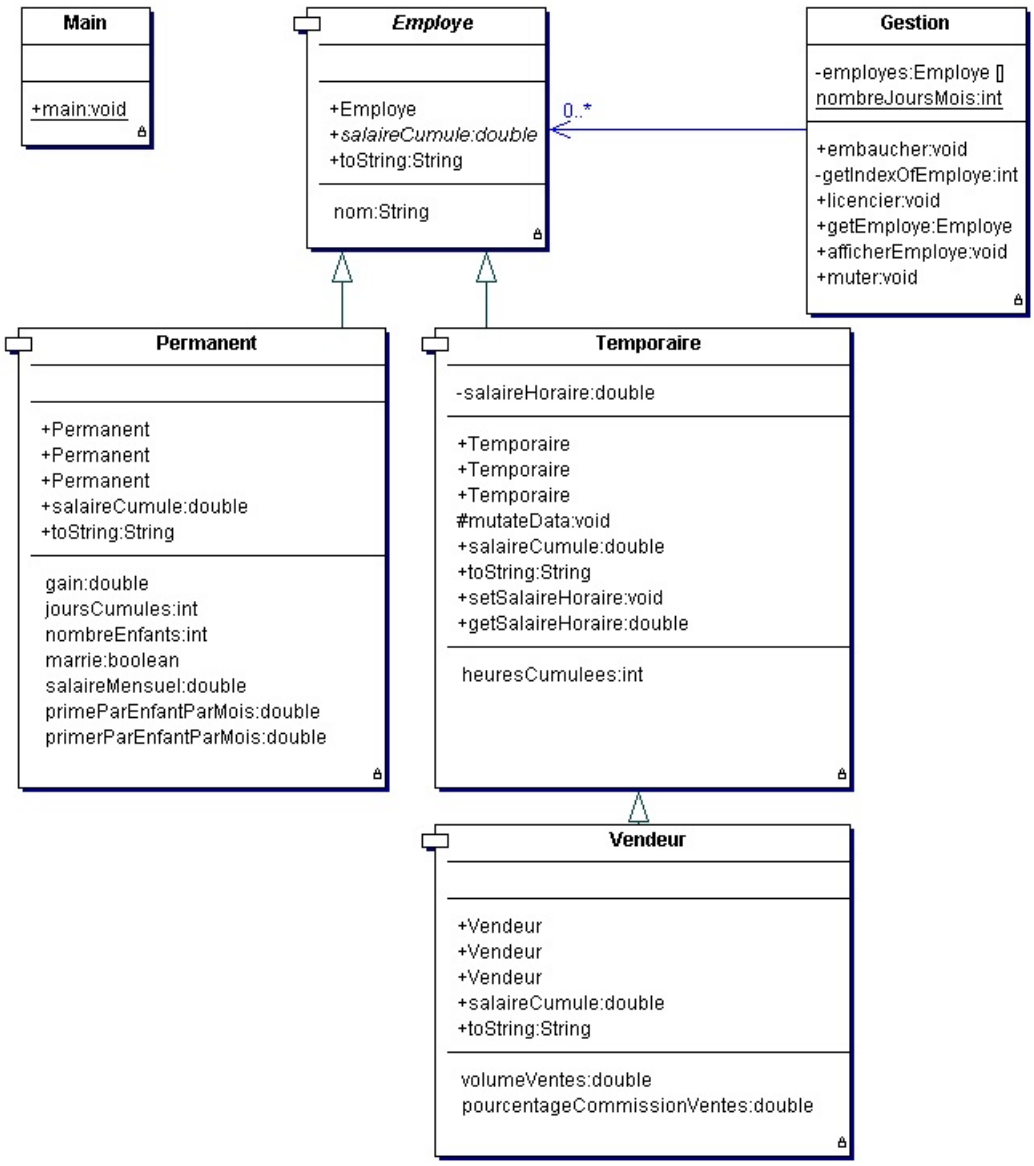
```
  /**
   * Affiche l'employ'e
   */
  public void afficherEmploye(String nom) {
    getEmploye(nom).affiche();
  }

  /**
   * employeNouvelleCategorie doit etre initialis'ee avec le nouveau type
   * d'Employe en utilisant son constructeur qui prend l'ancien employ'e
   *comme parametre. Regardez le commentaire de la classe Gestion.
   */
  public void muter(Employe employeNouvelleCategorie) {
    int index = getIndexOfEmploye(employeNouvelleCategorie.getNom());

    if (index == -1) {
      System.out.println("Il n'y a pas d'employ'e nomm'e " +
             employeNouvelleCategorie.getNom());
    }

    employes[index] = employeNouvelleCategorie;
  }
}

/**
 * La classe de base pour tous les types d'employ'es
 */
abstract class Employe {

  // Nom de l'employ'e
  // Son statut est donn'e par le type de classe
  private String nom;

  // Constructeur de base
  public Employe(String nom) {
    this.nom = nom;
  }

  /**
   * Chaque type d'employ'e doit retourner le salaire cumul'e en CHF
   */
  abstract public double salaireCumule();

  /**
   * Change le nom d'un employ'e
   */
  public void setNom(String nom) {
    this.nom = nom;
  }

  /**
   * Retourne le nom d'un employ'e
   */
  public String getNom() {
    return nom;
  }

  public void affiche() {
    System.out.println(" nom de l'employ'e=" + nom);
    System.out.println(" salaire cumul'e=" + salaireCumule());
  }
}

/**
 * C'est un type d'employ'e qui a un salaire mensuel fixe
 */
class Permanent extends Employe {

  private int nombreEnfants;
  private boolean marie;

  private double salaireMensuel;
  private double primeParEnfantParMois;

  private int joursCumules;
```

```java
/**
 * Constructeur de base
 */
public Permanent(String nom, int nombreEnfants, boolean marie,
         double primeParEnfantParMois, double salaireMensuel,
         int joursCumules) {
  super(nom);
  this.nombreEnfants = nombreEnfants;
  this.marie = marie;
  this.primeParEnfantParMois = primeParEnfantParMois;
  this.salaireMensuel = salaireMensuel;
  this.joursCumules = joursCumules;
}


/**
 * Constructeur utilis'e pour la mutation. On doit lui fournir l'employ'e
 * avant la mutation, ie, "ancien".
 * On prend son nom et on utilise son salaireCumul'e pour le convertir de
 * façon a` etre compatible avec sa nouvelle categorie
 */
public Permanent(Employe ancien, int nombreEnfants, boolean marie,
         double primeParEnfantParMois, double salaireMensuel) {
  // this permet ici d'invoquer le constructeur de la classe
  // dont la liste de paramètres correspond a` celle fournie (ici ce sera
  // le constructeur pr'ec'edent.
  // C'est une nouvelle instruction que vous d'ecouvrez ici.
  // Vous pouvez en fait vous en passer, mais ceci impliquerait de dupliquer
  // toutes les lignes de code li'ees a` l'initialisation des attributs
  // nombreEnfant, marie,primeParEnfantParMois et salaireMensuel.
  this(ancien.getNom(), nombreEnfants, marie, primeParEnfantParMois,
   salaireMensuel,0);
  joursCumules = (int) (ancien.salaireCumule() *
            (Gestion.nombreJoursMois / getGain()));
}

/**
 * Retourne le salaire cumul'e en CHF
 */
public double salaireCumule() {
  return (getGain() * joursCumules / (double) Gestion.nombreJoursMois);
}

/**
 * Retourne le salaire mensuel fixe (contenant aussi les primes)
 */
private double getGain() {
  double gain = salaireMensuel;
  if (marie && nombreEnfants > 0) {
    gain += primeParEnfantParMois * nombreEnfants;
  }
  return gain;
}

public void affiche() {
  super.affiche();
  System.out.println(" nombre d'enfants=" + nombreEnfants);
  System.out.println(" mari'e=" + marie);
  System.out.println(" salaire mensuel=" + salaireMensuel);
  System.out.println(" prime/enfant/mois=" + primeParEnfantParMois);
}

public void setJoursCumules(int joursCumules) {
  this.joursCumules = joursCumules;
}

public int getJoursCumules() {
  return joursCumules;
}

public void setNombreEnfants(int nombreEnfants) {
  this.nombreEnfants = nombreEnfants;
}

public int getNombreEnfants() {
```

```
    return nombreEnfants;
  }

  public void setMarie(boolean marie) {
    this.marie = marie;
  }

  public boolean estMarie() {
    return marie;
  }

  public void setSalaireMensuel(double salaireMensuel) {
    this.salaireMensuel = salaireMensuel;
  }

  public double getSalaireMensuel() {
    return salaireMensuel;
  }

  public void setPrimeParEnfantParMois(double primerParEnfantParMois) {
    if(marie) {
      this.primeParEnfantParMois = primerParEnfantParMois;
    } else {
      this.primeParEnfantParMois = 0;
    }
  }

  public double getPrimerParEnfantParMois() {
    return primeParEnfantParMois;
  }

}


/**
 * C'est un type d'employ'e pay'e a` l'heure
 */
class Temporaire extends Employe {

  private double salaireHoraire;

  private int heuresCumulees;

  /**
   * Constructeur de base
   */
  public Temporaire(String nom, double salaireHoraire, int heuresCumulees) {
    super(nom);
    this.salaireHoraire = salaireHoraire;
    this.heuresCumulees = heuresCumulees;
  }

  /**
   * Constructeur de base
   */
  public Temporaire(String nom, double salaireHoraire) {
    this(nom, salaireHoraire, 0);
  }

  /**
   * Constructeur utilis'e par la mutation. On doit lui fournir l'employ'e
   * avant la mutation, ie, "ancien".
   * On prend son nom et on utilise son salaireCumul'e pour le convertir de
   * façon a` etre compatible avec sa nouvelle categorie
   */
  public Temporaire(Employe ancien, double salaireHoraire) {
    // si vous ne voyez pas ce qe veut dire ce this
    // retourner au commentaire associ'e au constructeur
    // de la classe Permanent.
    this(ancien.getNom(), salaireHoraire);
    muterDonnees(ancien);
  }

  /**
   * Cette m'ethode effectue la mutation depuis une autre cat'egorie d'employ'es
   * en mettant a` jour l'attribut <heuresCumulees> en fonction du
```

```
    * <salaireCumule> de l'Employe.
    * @param ancien  instance de l'ancien type de l'employe
    */
  protected void muterDonnees(Employe ancien) {
    setHeuresCumulees((int) (ancien.salaireCumule() / salaireHoraire));
  }

  /**
   * Retourne le salaire cumul'e en CHF
   */
  public double salaireCumule() {
    return heuresCumulees * salaireHoraire;
  }

  public void affiche() {
    super.affiche();
    System.out.println(" salaire horaire=" + salaireHoraire);
    System.out.println(" heures cumul'ees=" + heuresCumulees);
  }

  public void setHeuresCumulees(int heuresCumulees) {
    this.heuresCumulees = heuresCumulees;
  }

  public int getHeuresCumulees() {
    return heuresCumulees;
  }

  public void setSalaireHoraire(int salaireHoraire) {
    this.salaireHoraire = salaireHoraire;
  }

  public double getSalaireHoraire() {
    return salaireHoraire;
  }
}


class Vendeur extends Temporaire {

  private double volumeVentes;
  private double pourcentageCommissionVentes;

  /**
   * Constructeur de base
   */
  public Vendeur(String nom, double salaireHoraire, double
         pourcentageCommissionVentes, int heuresCumulees,
         int volumeVentes) {
    super(nom, salaireHoraire, heuresCumulees);
    this.pourcentageCommissionVentes = pourcentageCommissionVentes;
    this.volumeVentes = volumeVentes;
  }

  /**
   * Constructeur de base
   */
  public Vendeur(String nom, double salaireHoraire,
         double pourcentageCommissionVentes) {
    // appel du constructeur de base
    this(nom, salaireHoraire, pourcentageCommissionVentes, 0, 0);
  }

  /**
   * Constructeur utilis'e par la mutation. On doit lui fournir l'employ'e
   * avant la mutation, ie, "ancien".
   * On prend son nom et on utilise son salaireCumul'e pour le convertir
   * de façon a` etre compatible avec sa nouvelle categorie
   */
  public Vendeur(Employe ancien, double salaireHoraire, double pourcentageCommissionVentes) {
    this(ancien.getNom(), salaireHoraire, pourcentageCommissionVentes, 0, 0);
    muterDonnees(ancien);
  }

  /**
   * Retourne le salaire cumul'e en CHF
```

```
   */
  public double salaireCumule() {
    return super.salaireCumule() + volumeVentes * pourcentageCommissionVentes;
  }

  public void affiche() {
    super.affiche();
    System.out.println( " volume de ventes=" + volumeVentes);
    System.out.println(" commission sur la vente=" +
               pourcentageCommissionVentes);
  }

  public void setVolumeVentes(double volumeVentes) {
    this.volumeVentes = volumeVentes;
  }

  public double getVolumeVentes() {
    return volumeVentes;
  }

  public void setPourcentageCommissionVentes(double pourcentageCommissionVentes) {
    this.pourcentageCommissionVentes = pourcentageCommissionVentes;
  }

  public double getPourcentageCommissionVentes() {
    return pourcentageCommissionVentes;
  }
}
```



Exercice 24 : Déménagement

```
// Hierarchie de classe pour les items
// (i.e. les choses que l'on peut mettre dans un carton)

abstract class Item {
    public abstract void print();
    public abstract boolean find(String ItemName);
}

// Un objet simple est un Item
class SimpleItem extends Item {

    private String name;

    public SimpleItem(String aName) {
        name = aName;
    }

    public void print(){
        System.out.println(name);
    }

    public boolean find(String ItemName){
        return (name.equals(ItemName));
    }

}

// et un carton est aussi un Item
// ici la subtilit'e est dans la structure r'ecursive de
// la classe Box: une Box est un Item et contient des Items
class Box extends Item {

    private Item content[];
    private int nbItems;
    private int number;

    public Box(int size, int aNumber){
        content = new Item[size];
        nbItems = -1;
        number = aNumber;
    }

    public void addItem(Item item){
        if (nbItems < content.length - 1) {
            ++nbItems;
            content[nbItems] = item;
        }
    }

    public int getNumber(){
        return number;
    }

    public void print(){
        for (int i = 0; i <= nbItems; ++i) {
            content[i].print();
        }
    }

    public boolean find (String ItemName){
        for (int i = 0; i <= nbItems; ++i) {
            if (content[i].find(ItemName))
                return true;
        }
        return false;
    }

}

// Le demenagement est un ensemble de cartons
public class Demenagement{

    private Box[] boxes;
```

```
    private int index;

    public Demenagement(int boxNumber){
        boxes = new Box[boxNumber];
        index = -1;
    }

    public void addBox(Box box){
        if (index < boxes.length - 1) {
            ++ index;
            boxes[index] = box;
        }
    }

    public void print(){
        System.out.println("Les objets de mon demenagement sont :");
        for (int i = 0; i <= index; ++i) {
            boxes[i].print();
        }
    }

    // la nature r'ecursive des cartons
    // fait qu'une recherche r'ecursive est
    // la solution la plus naturelle
    public int find(String itemName){
        for (int i = 0; i <= index; ++i) {
            if (boxes[i].find(itemName))
                return boxes[i].getNumber();
        }
        return -1;
    }

    public static void main(String[] args) {
        //On cr'ee un d'em'enagement constitu'e de 2 cartons principaux
        Demenagement demenagement = new Demenagement(2);

        //On cr'ee et remplis ensuite 3 cartons
        Box box1 = new Box(1,1);
        box1.addItem(new SimpleItem("ciseaux"));
        Box box2 = new Box(1,2);
        box2.addItem(new SimpleItem("livre"));
        Box box3 = new Box(2,3);
        box3.addItem(new SimpleItem("boussole"));

        //On ajoute ensuite un autre carton a` ce dernier qui contient un objet "'echarpe"
        Box box4 = new Box(1,4);
        box4.addItem(new SimpleItem("echarpe"));
        box3.addItem(box4);

        //On ajoute les trois cartons au premier carton du d'em'enagement
        Box box5 = new Box(3,5);
        box5.addItem(box1);
        box5.addItem(box2);
        box5.addItem(box3);

        //On ajoute un carton contenant 3 objets au d'em'enagement
        Box box6 = new Box(3,6);
        box6.addItem(new SimpleItem("crayons"));
        box6.addItem(new SimpleItem("stylos"));
        box6.addItem(new SimpleItem("gomme"));


        //On ajoute les deux cartons les plus externes au d'em'enagement
        demenagement.addBox(box5);
        demenagement.addBox(box6);

        //On imprime tout le contenu du d'em'enagement
        demenagement.print();

        //On imprime le num'ero du carton le plus externe contenant l'objet "'echarpe"
        System.out.println("L''echarpe est dans le carton num'ero " + demenagement.find("echarpe"));
    }
}
```

Exercice 25 : Expressions arithmétiques

Discussion

Le but de cet exercice est de faire la comparaison entre une approche orientée objet et une approche procédurale pour un problème donné. Dans l'approche orientée objet, données et méthodes relatives à une expression sont encapsulés dans l'objet représentant l'expression. Le polymorphisme permet de générer des expressions à partir d'autres expressions sans que cela ne nécessite le recours à des tests spécifiques. Les principaux avantages de la solution orientée objet sont :

Simplicité

La variante OO est écrite beaucoup plus clairement et simplement que sa contrepartie procédurale. Pour vous en convaincre, comparez le `main` des deux variantes.

Facilité de maintenance et de modification

Il est beaucoup plus facile d'ajouter un nouveau type d'expression dans la solution orientée objet. Il suffit de créer une nouvelle classe et de la doter des méthodes nécessaires.

Ces avantages font en fait la force de l'approche orientée-objet. Cela permet de créer du code évolutif plus facile à lire, à modifier et à maintenir.

Sources

Voici la version orientée-objet du programme original de Johnny (programme fourni `Arith.java`):

```
/** Interface donnant un type commun a toutes les expressions */
abstract class Expression {

    public abstract int evaluate();

}

/** Une classe pour les nombres */
class Number extends Expression {
    private int value;

    public Number(int value) {
        this.value = value;
    }

    public int evaluate() {
        return value;
    }
}

/** Une classe abstraite pour les operateurs binaires */
abstract class BinOp extends Expression {

    protected Expression leftOp;
    protected Expression rightOp;

    public BinOp(Expression leftOp, Expression rightOp) {
        this.leftOp = leftOp;
        this.rightOp = rightOp;
    }

}

/** Une classe pour les sommes */
class Sum extends BinOp {

    public Sum(Expression leftOp, Expression rightOp) {
        super(leftOp, rightOp);
    }

    public int evaluate() {
        return leftOp.evaluate() + rightOp.evaluate();
    }

}

/** Une classe pour les produits */
class Prod extends BinOp {

    public Prod(Expression leftOp, Expression rightOp) {
        super(leftOp, rightOp);
```

```
    }

    public int evaluate() {
        return leftOp.evaluate() * rightOp.evaluate();
    }

}

/** Classe principale */
class ArithOOP1Bis {

    public static void main(String [] args) {
        //construit l'expression 3 + 2 * 5
        Expression term = new Sum(
            new Number(3),
            new Prod(
                new Number(2),
                new Number(5)));

        System.out.println(term.evaluate());
    }

}
```

Voici maintenant le programme de Johnny doté des nouvelles facilités :

```
/** Structure de donnee pour une Expression */
class Expression {
    public int type;
    public int value;
    public Expression leftOp;
    public Expression rightOp;

    public Expression(int type, int value, Expression leftOp, Expression rightOp) {
        this.type = type;
        this.value = value;
        this.leftOp = leftOp;
        this.rightOp = rightOp;
    }
}

/** Classe principale */
class Arith {

    /** Constantes pour les types*/
    public static final int TYPE_NUMBER = 1;
    public static final int TYPE_SUM = 2;
    public static final int TYPE_PROD = 3;
    public static final int TYPE_MODULO = 4;

    public static void main(String [] args) {
        //Construit l'expression (3 + 2 * 5 ) % 7
        Expression term = new Expression(TYPE_MODULO, 0,
            new Expression(TYPE_SUM, 0,
                new Expression(TYPE_NUMBER, 3, null, null),
                new Expression(TYPE_PROD, 0,
                    new Expression(TYPE_NUMBER, 2, null, null),
                    new Expression(TYPE_NUMBER, 5, null, null))),
            new Expression(TYPE_NUMBER, 7, null, null));

        System.out.println(print(term) + " s'evalue " + evaluate(term));
    }

    /** Evaluation recursive d'une expression */
    public static int evaluate(Expression term) {
        switch (term.type) {
            case TYPE_NUMBER:
                return term.value;
            case TYPE_SUM:
                return evaluate(term.leftOp) + evaluate(term.rightOp);
            case TYPE_PROD:
                return evaluate(term.leftOp) * evaluate(term.rightOp);
            case TYPE_MODULO:
                return evaluate(term.leftOp) % evaluate(term.rightOp);
            default:
                return 0;   //erreur, ne devrait jamais se produire
```

```
        }
    }

    /** Retourne une String decrivant l'expression */
    public static String print(Expression term) {
        switch (term.type) {
            case TYPE_NUMBER:
                return " " + term.value + " ";
            case TYPE_SUM:
                return print(term.leftOp) + "+" + print(term.rightOp);
            case TYPE_PROD:
                return print(term.leftOp) + "*" + print(term.rightOp);
            case TYPE_MODULO:
                return print(term.leftOp) + "%" + print(term.rightOp);
            default:
                return "INVALID";
        }
    }

}
```

Et voici enfin la variante orientée objet de ce programme :

```
/** Interface donnant un type commun a toutes les expressions*/
abstract class Expression {

    public abstract int evaluate();
    public abstract String print();

}

/** Une classe pour les nombres */
class Number extends Expression {
    private int value;

    public Number(int value) {
        this.value = value;
    }

    public int evaluate() {
        return value;
    }

    public String print() {
        return " " + value + " ";
    }

}

/** Une classe abstraite pour les operateurs binaires */
abstract class BinOp extends Expression {

    protected Expression leftOp;
    protected Expression rightOp;

    public BinOp(Expression leftOp, Expression rightOp) {
        this.leftOp = leftOp;
        this.rightOp = rightOp;
    }

}


/** Une classe pour les sommes */
class Sum extends BinOp {

    public Sum(Expression leftOp, Expression rightOp) {
        super(leftOp, rightOp);
    }

    public int evaluate() {
        return leftOp.evaluate() + rightOp.evaluate();
    }

    public String print() {
        return leftOp.print() + "+" + rightOp.print();
    }
```

```
}

/** Une classe pour les produits */
class Prod extends BinOp {

    public Prod(Expression leftOp, Expression rightOp) {
        super(leftOp, rightOp);
    }

    public int evaluate() {
        return leftOp.evaluate() * rightOp.evaluate();
    }

    public String print() {
        return leftOp.print() + "*" + rightOp.print();
    }
}

/** Une classe pour le modulo */
class Modulo extends BinOp {

    public Modulo(Expression leftOp, Expression rightOp) {
        super(leftOp, rightOp);
    }

    public int evaluate() {
        return leftOp.evaluate() % rightOp.evaluate();
    }

    public String print() {
        return leftOp.print() + "%" + rightOp.print();
    }
}



/** Classe principale */
class ArithOOP2Bis {

    public static void main(String [] args) {
        //construit l'expression (3 + 2 * 5) % 7
        Expression term = new Modulo(
            new Sum(
                new Number(3),
                new Prod(
                    new Number(2),
                    new Number(5))),
            new Number(7));

        System.out.println(term.print() + " s'evalue " + term.evaluate());
    }

}
```
