Correction TD n°2

Java Avancé

—M1 Apprentissage—

Redéfinitions, late binding, surcharge etc.

▸ Exercice 1. Redéfinition

```
class Mere {
  protected int meth() {
    return 42;
  }
  public void printMeth() {
    System.out.println(meth());
  }
}
class Fille extends Mere {
  public int meth() {
    return 24;
  }
}
class Main {
  public static void main(String[] args) {
    Mere mere = new Mere();
    System.out.println(mere.meth());
    mere.printMeth();

    Fille fille = new Fille();
    System.out.println(fille.meth());
    fille.printMeth();

    Mere mereFille = new Fille();
    System.out.println(mereFille.meth());
    mereFille.printMeth();
  }
}
```

1. Qu'affiche le main et pourquoi ?
2. S'il est dans `Fille`, à combien de méthodes `meth()` un objet de type `Fille` à accès (et comment il y accède) ? Et s'il est dans `Main` ?
3. Quel est le comportement si les méthodes `meth()` sont statiques ?
4. Et si meth sont maintenant des champs ? Pourquoi ?

```
class Mere {
  protected int meth=42;
  public void printMeth() {
    System.out.println(meth);
  }
}
class Fille extends Mere {
  public int meth = 24;
```

```java
}
class Main {
  public static void main(String[] args) {
    Mere mere = new Mere();
    System.out.println(mere.meth);
    mere.printMeth();

    Fille fille = new Fille();
    System.out.println(fille.meth);
    fille.printMeth();

    Mere mereFille = new Fille();
    System.out.println(mereFille.meth);
    mereFille.printMeth();
  }
}
```

---

1. — 42
   — 42
   — 24
   — 24 (appel meth de fille car a été redefini)
   — 24 (appel sur le "vrai" type de merefille, pas sur celui de sa variable de stockage. Meme principe que le toString redéfini de point...)
   — 24
2. 2 (meth() et super.meth()). 1 seule, pas de super !
3. — 42
   — 42
   — 24
   — 42 (appel du meth() en mode statique dans la classe)
   — 42 (appel selon le type de la variable. A ne pas faire normalement.)
   — 42
4. — 42
   — 42
   — 24
   — 42 (appel au CHAMP meth, pas de redefinition)
   — 42 il n'y a pas de résolution sur le type réel comme pour un appel de méthode, ici on demande explicitemenet le champs x d'un objet A (une bonne raison de plus pour avoir que des champs privés avec des accesseurs)
   — 42 idem que plus haut : la méthode est pas redefinie dans B donc c'est celle de A qui est appellée, dans A le seul champs x existant est celui de A

▸ Exercice 2. Redéfinition - surcharges

```java
class Mere {
  public void a() {System.out.println("Mere_a"); }
  void b(Fille fille) {System.out.println("Mere_b(Fille)");}

  void c() {System.out.println("Mere_c");}
```

```
6   void c(Mere mere) {System.out.println("Mere_c(Mere)"); }
7
8   static void d() {System.out.println("static Mere_d");}
9
10  protected void e() {System.out.println("Mere_e");}
11  private void f() {System.out.println("Mere_f");}
12  public void printF() { f(); }
13
14  Object g() {System.out.println("Mere_g"); return 2;}
15  int h() {System.out.println("Mere_h"); return 2;}
16  void i() {System.out.println("Mere_i");}
17
18  void j() throws Exception {System.out.println("Mere_j"); }
19  void k() throws IOException {System.out.println("Mere_k"); }
20  void l() throws Exception {System.out.println("Mere_l"); }
21  void m() throws Exception, ArrayIndexOutOfBoundsException {System.out.println("Mere_m"); }
22 }
23 class Fille extends Mere{
24  void miage() {System.out.println("Miage");}
25
26  public void a() {System.out.println("Fille_a"); }
27
28  protected void b(Fille fille) {System.out.println("Fille_b(Fille)");}
29
30  public void c(Mere mere) {System.out.println("Fille_c(Mere)");}
31  void c(Fille b) {System.out.println("Fille_c(Fille)"); }
32
33  static void d() {System.out.println("static Fille_d");}
34  static void d(Mere mere) {System.out.println("Fille_d(Mere)");}
35
36  private void e() {System.out.println("Fille_e");}
37  protected void f() {System.out.println("Fille_f");}
38
39  String g() {System.out.println("Fille_g"); return "c";}
40  char h() {System.out.println("Fille_h"); return 'c';}
41  int i() {System.out.println("Fille_i"); return 3; }
42
43  void j() throws IOException {System.out.println("Fille_j"); }
44  void k() throws Exception {System.out.println("Fille_k"); }
45  void l() {System.out.println("Fille_l");}
46  void m() throws IOException, IllegalArgumentException {System.out.println("Fille_m"); }
47 }
48 public class Main {
49  public static void main(String[] args) throws Exception {
50    Mere mere=new Mere();
51    Mere mereFille=new Fille();
52    Fille fille=new Fille();
53
54    mere.miage();
55    fille.miage();
56    mereFille.miage();
57    ((Fille)mereFille).miage();
58
59    mere.a();
60    mereFille.a();
61    fille.a();
62    ((Mere)mereFille).a();
63    mereFille.b(null);
64
65    mereFille.c();
66    mereFille.c(mere);
67    mereFille.c(mereFille);
68    mereFille.c(fille);
69    fille.c(fille);
70
71    mere.d();
72    mereFille.d();
```

```

    mere.printF();
    mereFille.printF();

    mereFille.j();
    mereFille.k();
    mereFille.l();
    mereFille.m();
  }
}
```

1. Quelles sont les erreurs de compilation et pourquoi ?
2. Retirer les méthodes provoquant les erreurs.
3. Rappeler ce qu'est une redéfinition et une surcharge, et indiquer où sont les surcharges et où sont les redéfinitions ici.
4. Expliquer chaque affichage.

---

1. — mere.miage() car pas de methode miage dans mere
   — mereFille.miage() car est stocké dans une variable de type Mere donc pas acces a miage
   — La méthode e() de Fille réduit la visiblité de la méthode e() de Mere : une redefinition ne peut pas réduire sa visibilité
   — La méthode h() de Fille retourne un type primitif différent que la méthode h() de Mere : si le type de retour est un type primitif, ils doivent être de même type
   — La méthode i() de Fille retourne un type alors que la méthode i() de Mere ne retourne rien : non compatible
   — La méthode k() de Fille lève une exception plus "large" que la méthode k() de Mere.
2. .
3. Reféfinition : override la même méthode de la classe mère (même signature) Surcharge : même nom de méthode, mais les paramètres de la méthode sont différents
   — Surcharges :
     — c(Mere) sur c() dans Mere
     — c(Mere) et c(Fille) dans Fille
     — d(Mere) sur d() dans Fille
   — Redefinitions
     — a de Fille sur a de Mere (signature exacte)
     — b de Fille sur b de Mere : visibilité dans Fille plus large (protected ¿ rien) : ok
     — pareil pour c(Mere) : public ¿ rien
     — Il n'y en a pas pour c(Mere) de Fille ni c() de Mere
     — Il n'y en a pas pour d, méthodes statics
     — f de Fille sur f de Mere : non, ne peut pas redefinir car ne "voit" pas de méthode f() Mere car private ! ! En profiter pour (re)parler du @Override, bien utile ici car compile sans (et peut donc croire qu'il y a redefinition alors que non)
     — g() de Fille sur g() de Mere : type de retour dans Mere plus "haut" que celui dans Fille
     — j() de Fille sur j() de Mere : throws dans Mere plus "haut" que celui de Fille
     — l() de Fille sur l() de Mere : throws dans Mere plus "haut" que le rien de Fille
     — m() de Fille sur m() de Mere : le throws Exception de Mere est plus "haut" que ceux de Fille

```java
    Mere mere=new Mere();
    Mere mereFille=new Fille();
    Fille fille=new Fille();

    mere.miage(); //compile pas car pas de methode miage visible dans mere
    fille.miage(); //ok ras
    mereFille.miage(); //compile pas car pas de methode miage visible dans mere et mereFille est stocké dans une Mere
    ((Fille)mereFille).miage(); //ok, on a cast, mereFille est maintenant une Fille

    mere.a(); //Mere_a, ras
    mereFille.a(); // Fille_a: vrai type de mereFille est fille, et a redefini
    fille.a(); //Fille_a: idem.
    ((Mere)mereFille).a(); //Fille_a: cherche le vrai type, cast n'y change rien.
    mereFille.b(null); //Fille_b(Fille): vrai type est Fille, cherche b dans Fille avec un objet.

    mereFille.c(); //Mere_c: vrai type est Fille, mais c pas redefini dans Fille.
    mereFille.c(mere); //Fille_c(Mere): vrai type est fille, c redefini pour Mere en argument.
    mereFille.c(mereFille); //Fille_c(Mere): premiere methode compatible lors de la resolution
    mereFille.c(fille); //Fille_c(Mere) premiere methode compatible lors de la resolution: la méthode c(Fille) n'est pas visible par mereFille qui est de type Mere car non definie dans Mere et mereFille est stockée dans un Mere. (se verrait avec @Override qui marche que pour Mere en arg)
    fille.c(fille); //Fille_c(fille) car fille est bien stocké dans un Fille

    mere.d(); //static Mere_d: (mais mauvaise maniere de faire)
    mereFille.d();//static Mere_d: appel sur le type de la variable de stockage

    mere.printF(); //Mere_f ras
    mereFille.printF(); //Mere_f: pas redef dans Fille + f() n'existe pas pour Fille (car f est private) ! Si f() de fille redefinissait, affiche Fille_f

    mereFille.j(); //Fille_j: vrai type est Fille, j redefini
    mereFille.k();  //Mere_k: supprimé car pas de compil dans pas redefini
    mereFille.l(); //Fille_l: comme j
    mereFille.m(); //Fille_m: comme j
```

▸ Exercice 3. Expressions arithmétiques

On cherche à évaluer une expression arithmétique simple, représenté par un arbre. On veut un type commun `Expr` représentant des expressions arithmétiques, pouvant être soit de valeur réelle (`Value`) soit une opération d'addition (`Add`), permettant l'addition entre deux expressions. On veut pouvoir évaluer la valeur d'une expression au moyen d'une méthode `eval()`.

Par exemple :

```java
Expr val = new Value(1337.0);
System.out.println(val.eval()); //affiche 1337.0
Expr add = new Add(new Value(327.0), val);
System.out.println(add.eval()); //affiche 1664.0
Expr e = new Add(new Value(350.0), add);
System.out.println(e.eval()); //affiche 2014.0
```

1. Écrire les types `Expr, Value, Add`, les méthodes `eval` et une classe `Main` avec un Main de test dans un même package.

2. Implémenter l'affichage d'une expression arithmétique non évaluée.
3. Ajouter l'opération de multiplication.
4. Ajouter l'opération de racine carrée. Pour l'affichage, vous utiliserez le symbole unicode `\u221a`. Pour l'évaluation, vous utiliserez la méthode `Math.sqrt()`.
5. Tester avec le test JUnit suivant : ExprTest. Ajouter ce fichier dans votre projet, au même niveau que vos autres fichiers source. Faire "fix project setup", "add JUnit 4 to the path" pour l'erreur d'eclipse situé au niveau de l'import de JUnit. Faire Run as Junit test.

```java
package Expr;

public interface Expr {
  public double eval();
}
```

```java
package Expr;

public class Add implements Expr {
  private final Expr left, right;

  public Add(Expr l, Expr r) {
    left=l;
    right=r;
  }

  @Override
  public double eval() {
    return left.eval() + right.eval();
  }

  @Override
  public String toString() {
    return left.toString() + '+' + right.toString();
  }

}
```

```java
package Expr;

public class Val implements Expr{
  private final double val;

  public Val(double v) {
    this.val = v;
  }

  @Override
  public String toString() {
    return String.valueOf(val);
  }

  @Override
  public double eval() {
    return val;
  }
}
```

```java
package Expr;

public class Sqrt implements Expr{
  private final Expr e;
  public Sqrt(Expr e) {
    this.e = e;
  }
  @Override
  public double eval() {
    return Math.sqrt(e.eval());
  }

  @Override
  public String toString() {
    return "\u221a("+e+")";
  }

}
```

```java
package Expr;

public class Pow implements Expr{
  private final Expr a, b;
  public Pow(Expr a, Expr b) {
    this.a = a;
    this.b = b;
  }

  @Override
  public double eval() {
    return Math.pow(a.eval(), b.eval());
  }

  @Override
  public String toString() {
    return a + "^" + b;
  }
}
```

```java
package Expr;

public class Main {
  public static void main(String[] args) {
    Expr val = new Val(1337.0);
    System.out.println(val.eval()); //affiche 1337.0
    Expr add = new Add(new Val(327.0), val);
    System.out.println(add.eval()); //affiche 1664.0
    Expr e = new Add(new Val(350.0), add);
    System.out.println(e.eval()); //affiche 2014.0
    System.out.println(val);
    System.out.println(add);
    System.out.println(e);
    Expr s = new Sqrt(new Val(64));
    System.out.println(s);
    System.out.println(s.eval());
    Expr p = new Pow(new Val(2), new Val(3));
    System.out.println(p);
    System.out.println(p.eval());
  }
}
```