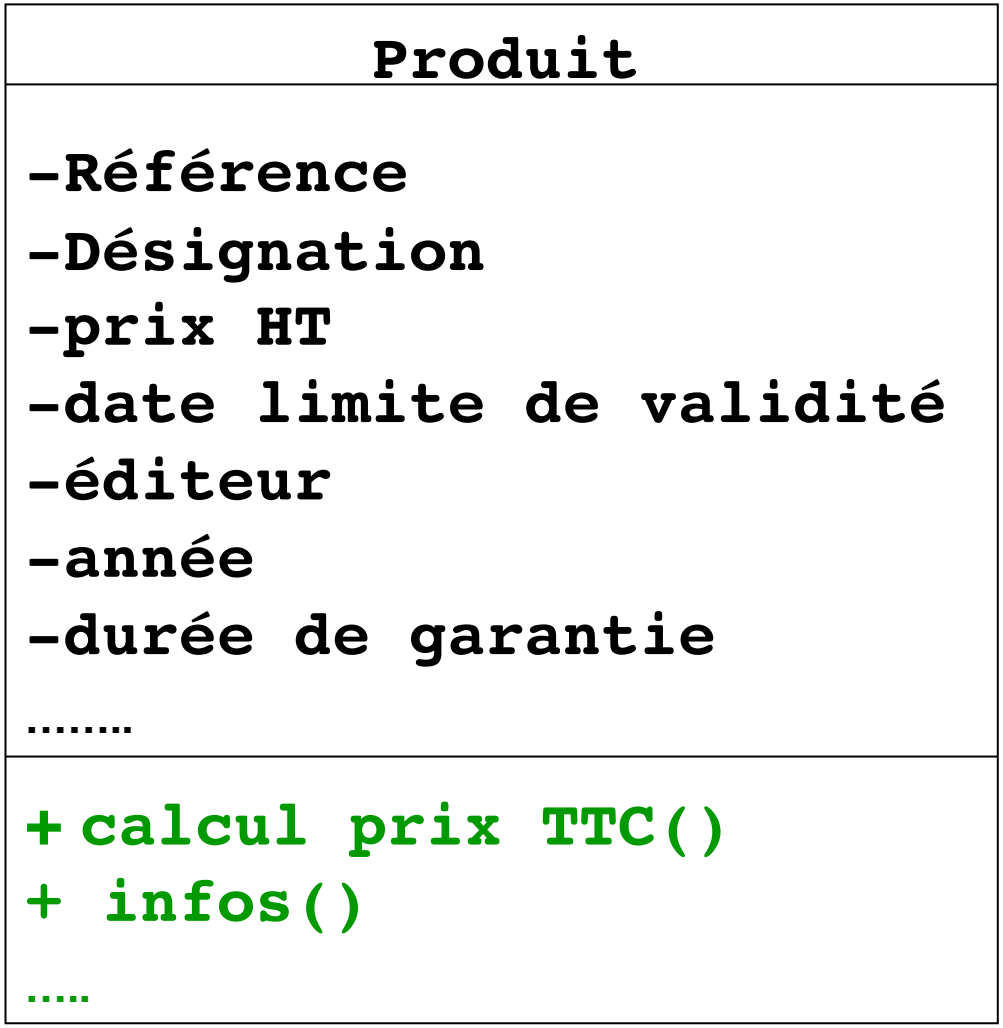
Spécialisation/généralisation
Héritage
Polymorphisme

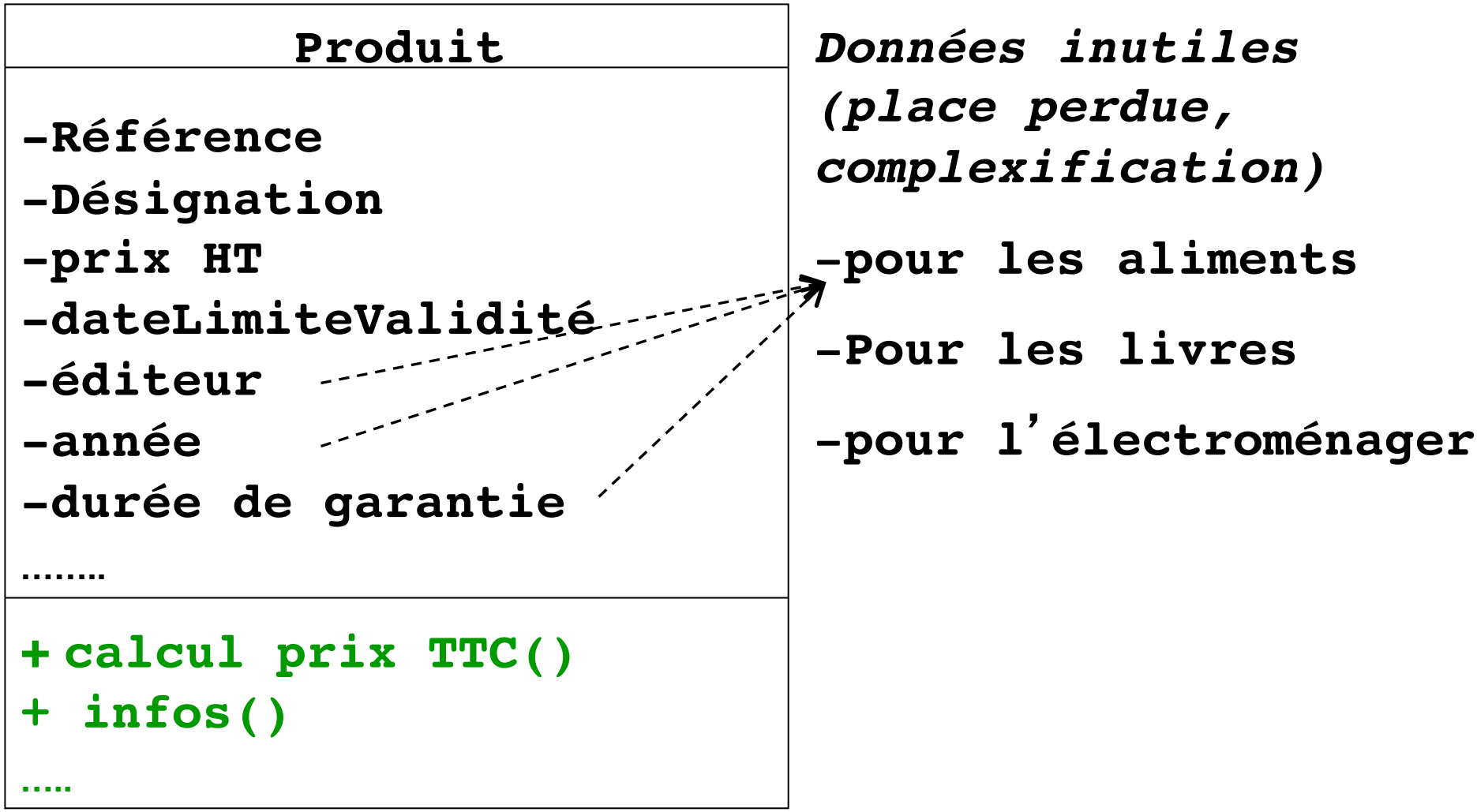
Un exemple

- Représentation de produits
  - Livres, aliments, articles d'électroménager
- Données spécifiques
  - Livre : éditeur, année
  - Aliments : date limite de validité
  - Electroménager : garantie
- Calculs de prix spécifiques
  - Livre et Aliment : TVA 5,5%
  - Aliment : réduction lorsqu'on approche de la date de péremption
  - Electroménager : TVA 19,6%

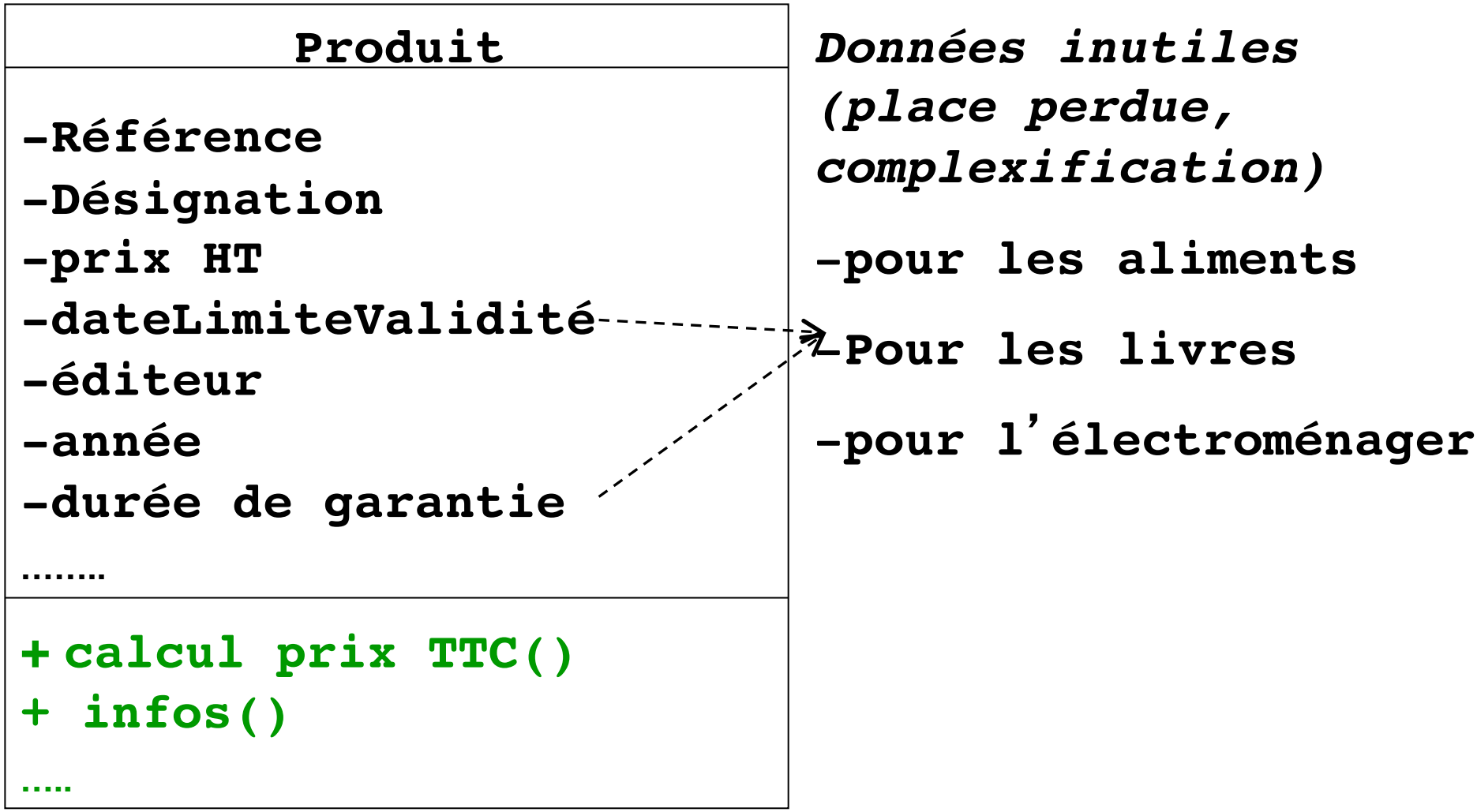
Solution 1 : une seule classe pour représenter tous les produits



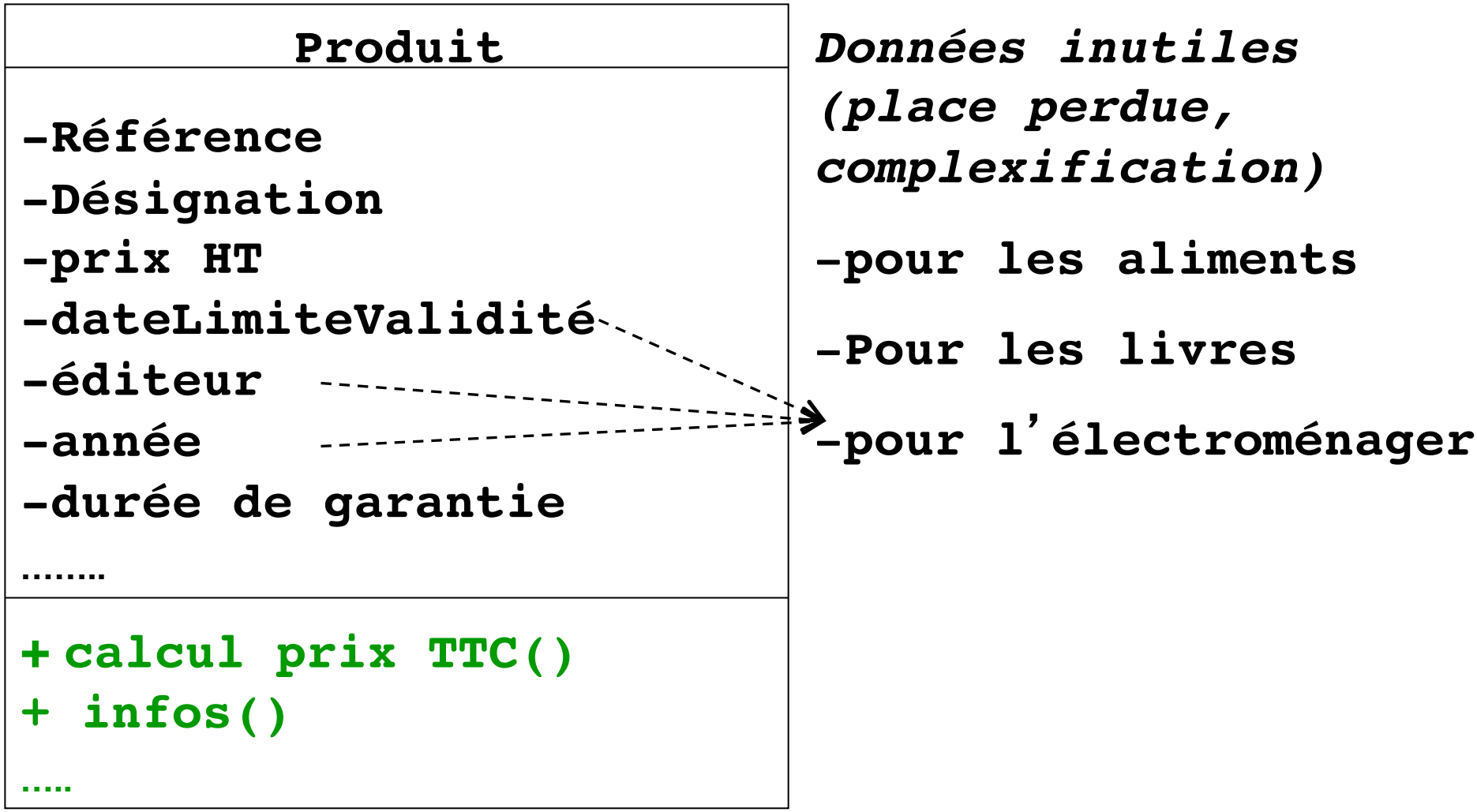
Solution 1 : une seule classe pour représenter tous les produits



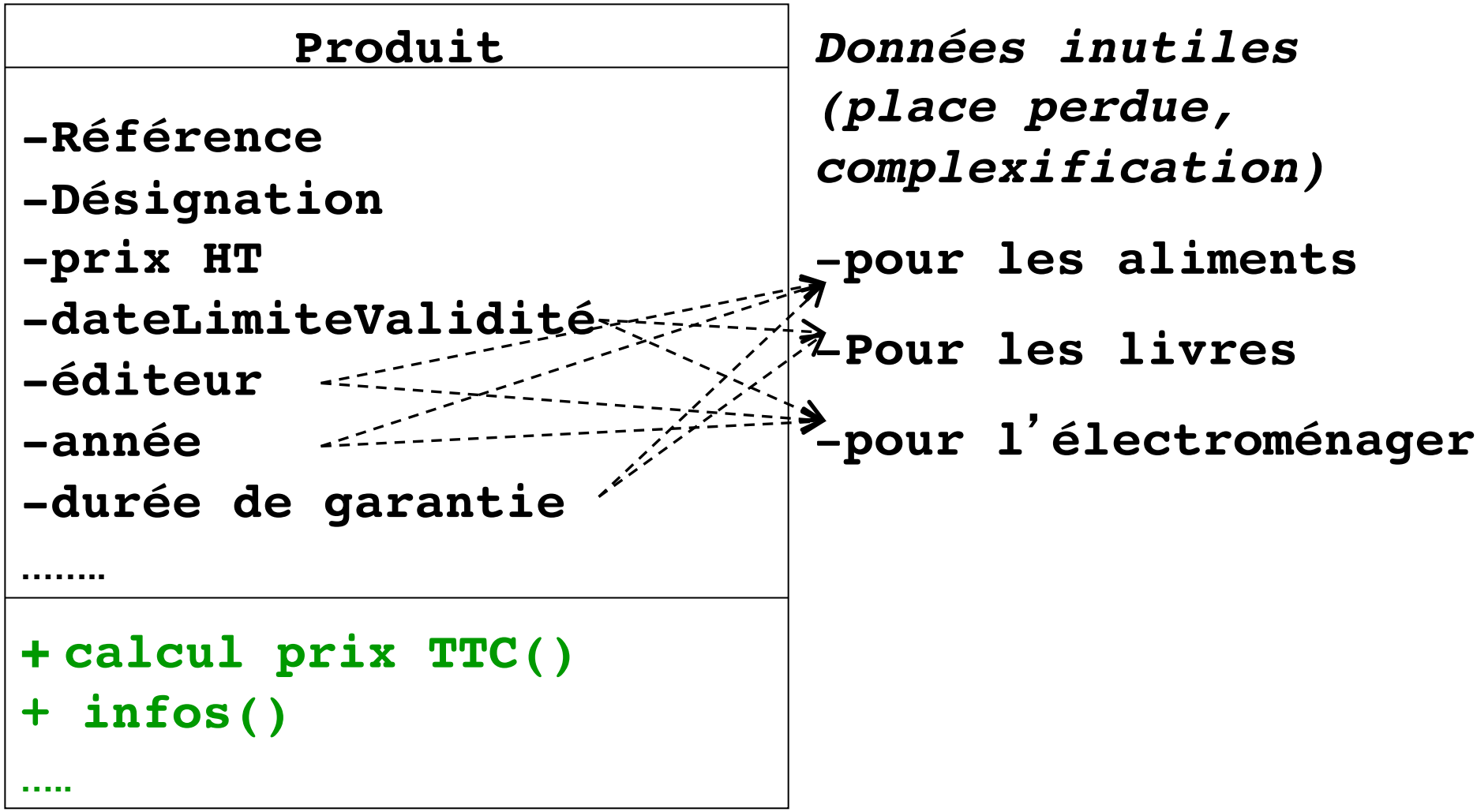
Solution 1 : une seule classe pour représenter tous les produits



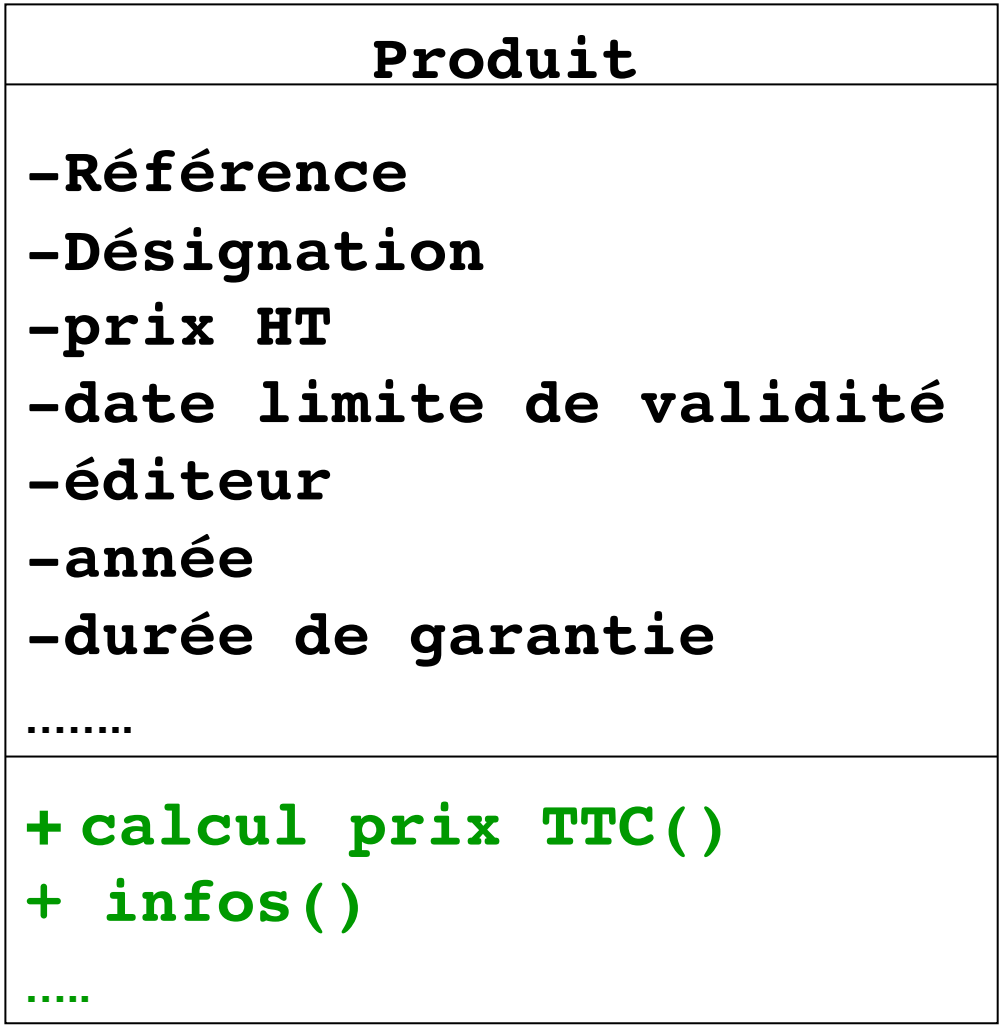
Solution 1 : une seule classe pour représenter tous les produits



Solution 1 : une seule classe pour représenter tous les produits



Solution 1 : une seule classe pour représenter tous les produits



Code complexe et non extensible dynamiquement à un nouveau type de produit

```
double prixTTC()

si(livre ou aliment) { … }
si(aliment){ … }
si (électro.){ … }

String infos()

Réf + désignation
si(livre) { … éditeur/an }
si(aliment){ … date limite}
si (électro.){ … garantie}
```

Solution 2 : une classe par produit

Aliment

Référence

Désignation

prix HT

date limite

prix TTC() 5,5%
et moins cher près

date limite

infos()
Réf + désignation

date limite .....

Livre

Référence

Désignation

prix HT

éditeur

année

prix TTC() 5,5%
infos()
Réf + désignation

éditeur/an .....

Electro

Référence

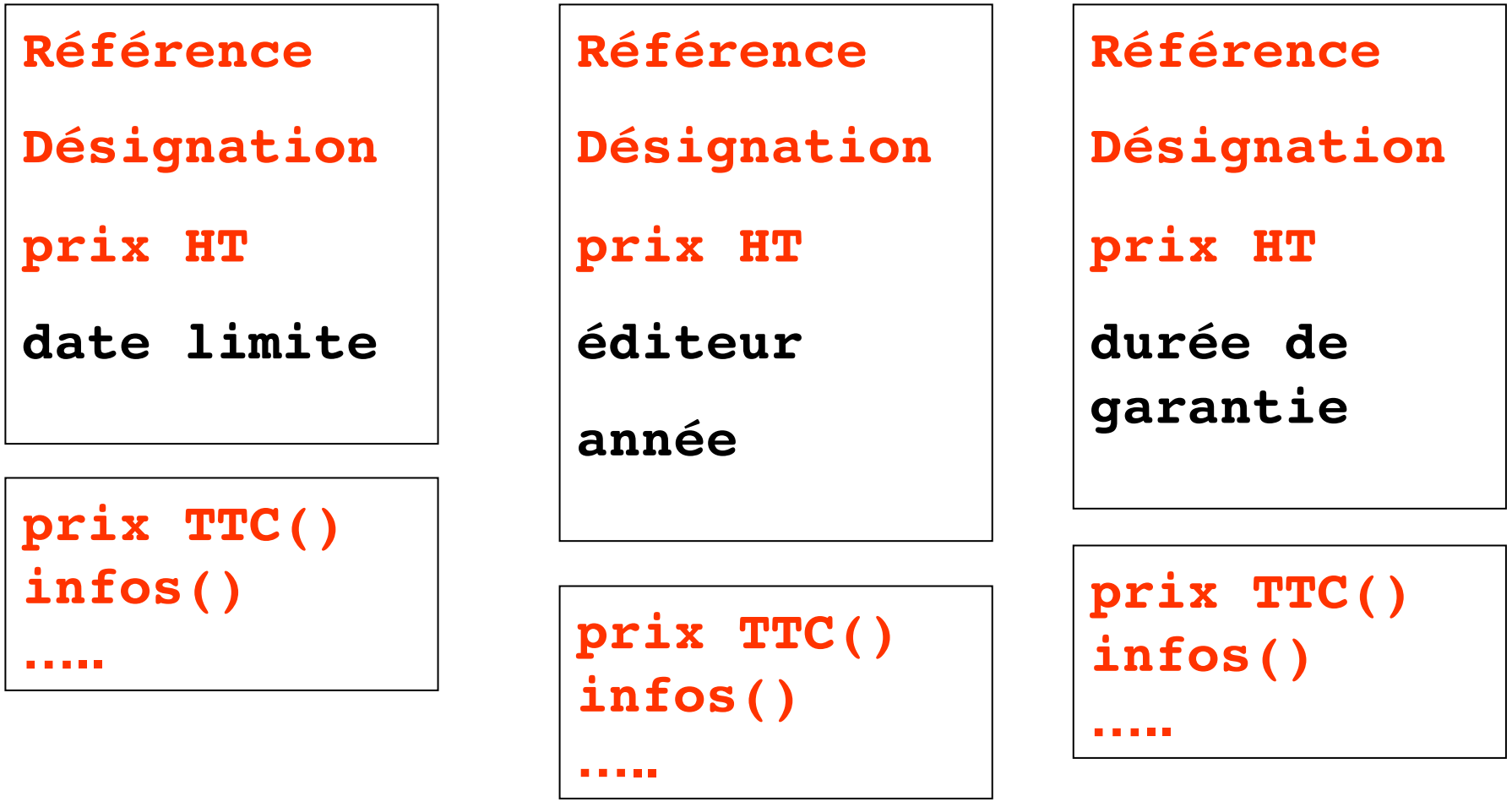
Désignation

prix HT

durée de garantie

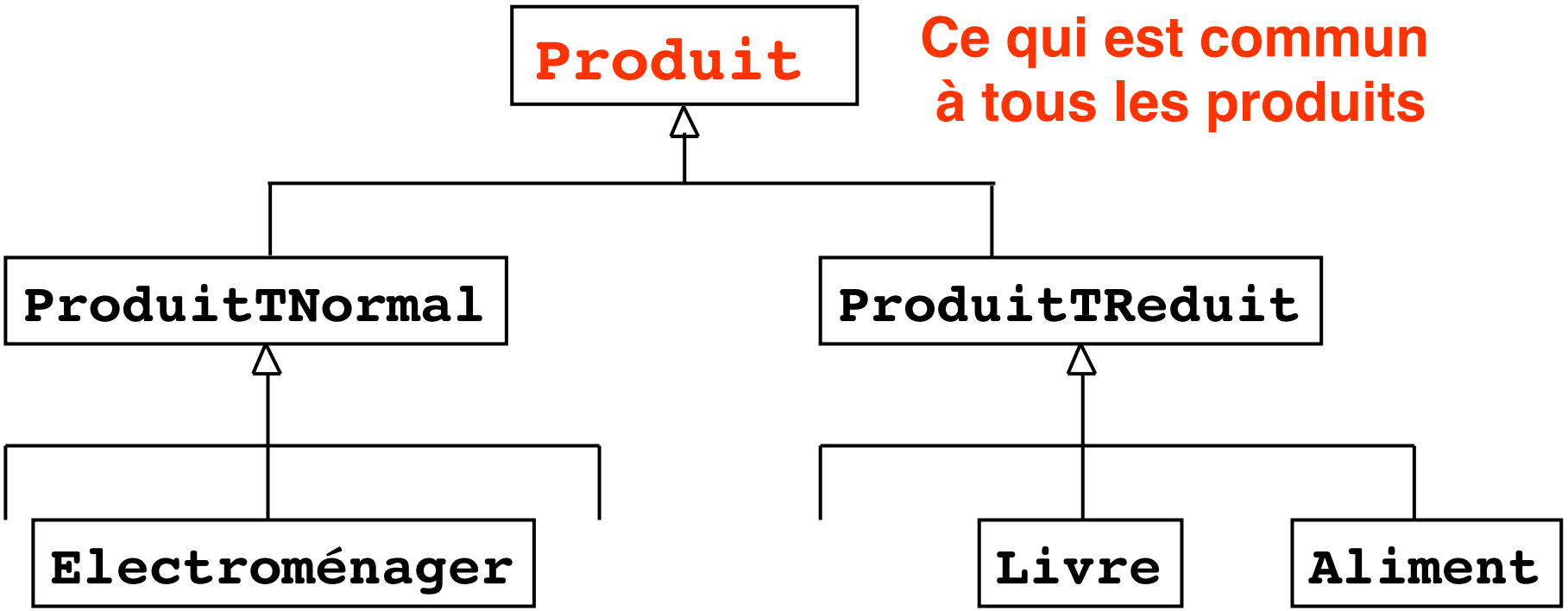
prix TTC() 19,6%
infos()
Réf + désignation

garantie .....

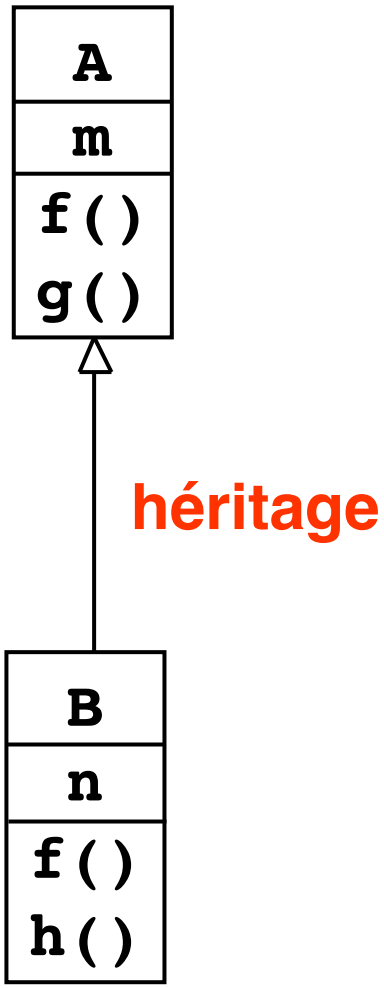


Répétitions -> Factoriser attributs et méthodes !

Solution 3 : organiser les classes en une hiérarchie d'héritage



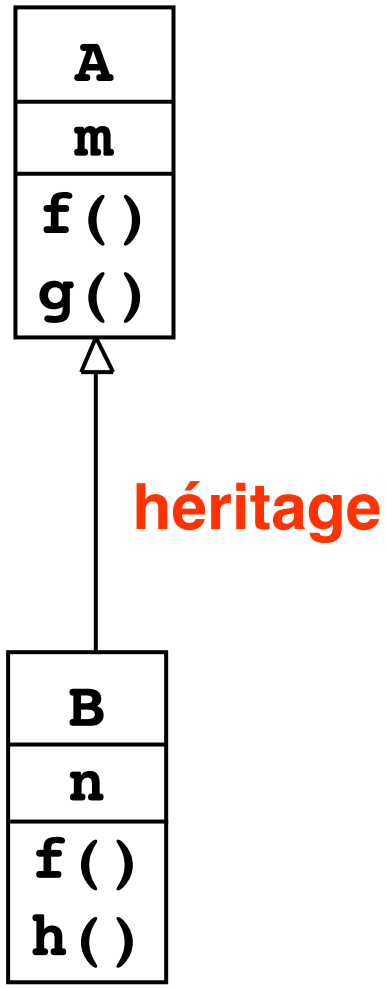
Héritage



B construite à partir de A en

- **ajoutant des attributs**
- **ajoutant des méthodes**
- **rédéfinissant des méthodes**

- **Une instance de B est composée des attributs m et n**
- **La redéfinition de f() dans B masque f() de A**

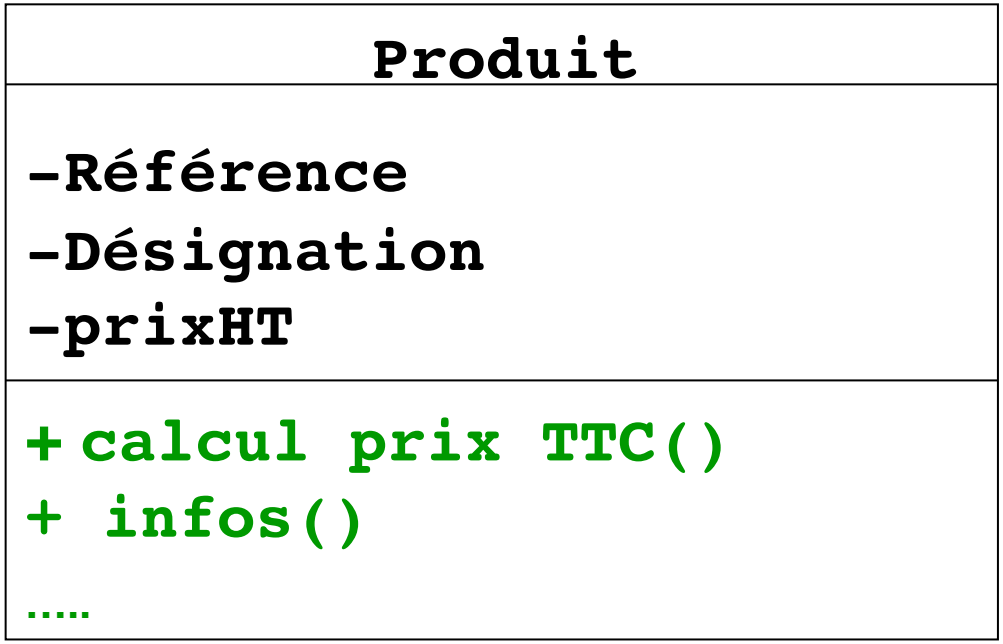



```
B b = new B();
b.f()      ;     // f de B
b.g()      ;     // g de A
b.h()      ;     // h de B
```

La classe Produit

première approche

factorise ce qui est commun

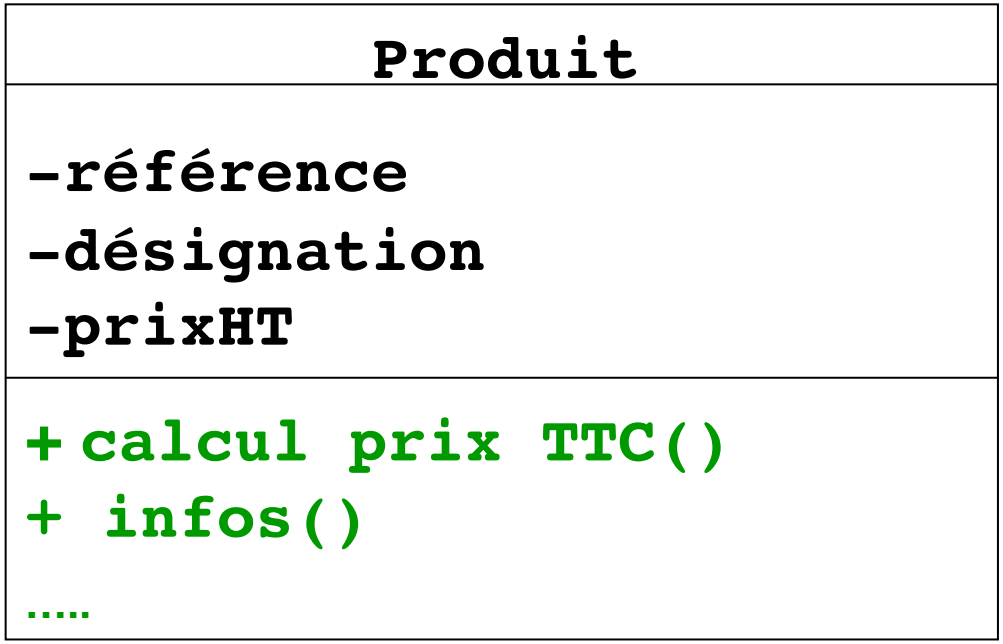



```
public class Produit

{ // attributs

private String reference;
private String designation;
private double prixHT;

// quelques méthodes

public double leprixTTC(){} //vide ..
public String infos(){..}

}
```

La classe Produit

première approche

Constructeur,
Accesseurs
aux attributs communs

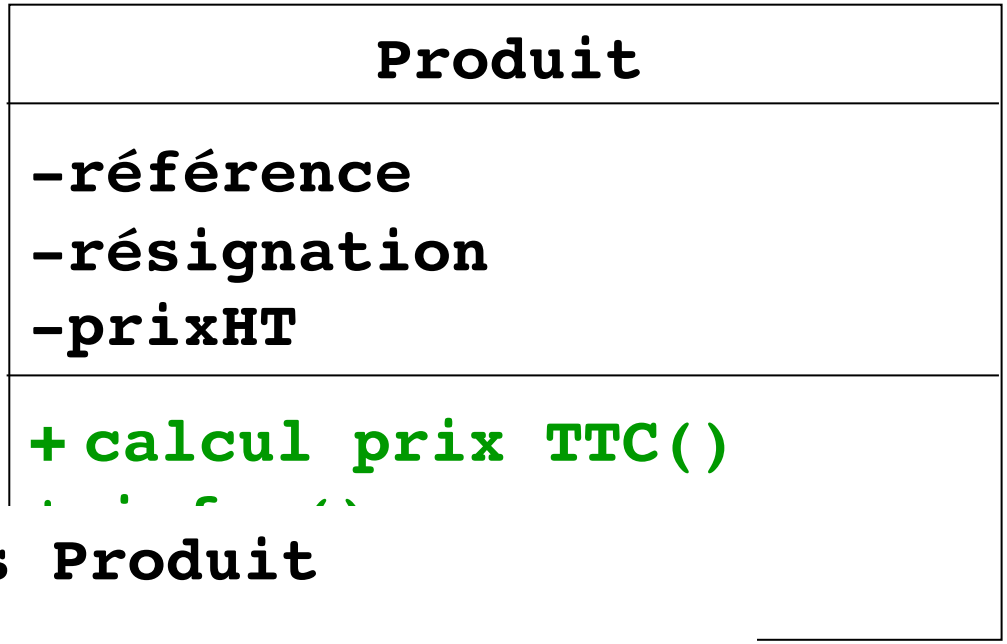



```
public class Produit

{ // attributs

private String reference;
private String designation;
private double prixHT;


//constructeur
public Produit(String r, String d, double p)
{reference=r; designation=d; prixHT=p;}

// quelques méthodes pour accéder aux attributs
public String getReference(){return reference;}
public void setReference(String r){reference=r;}
}
```

La classe Produit

n'est pas instanciable !

La méthode leprixTTC()
a un comportement inconnu




```
abstract public class Produit

{ // attributs

private String reference;
….

// quelques méthodes
…….

abstract public double leprixTTC();
public String infos(){..//ref+des+prix}

}
```

- **Calcul du prix TTC : méthode abstraite**

```
public abstract double leprixTTC();
```

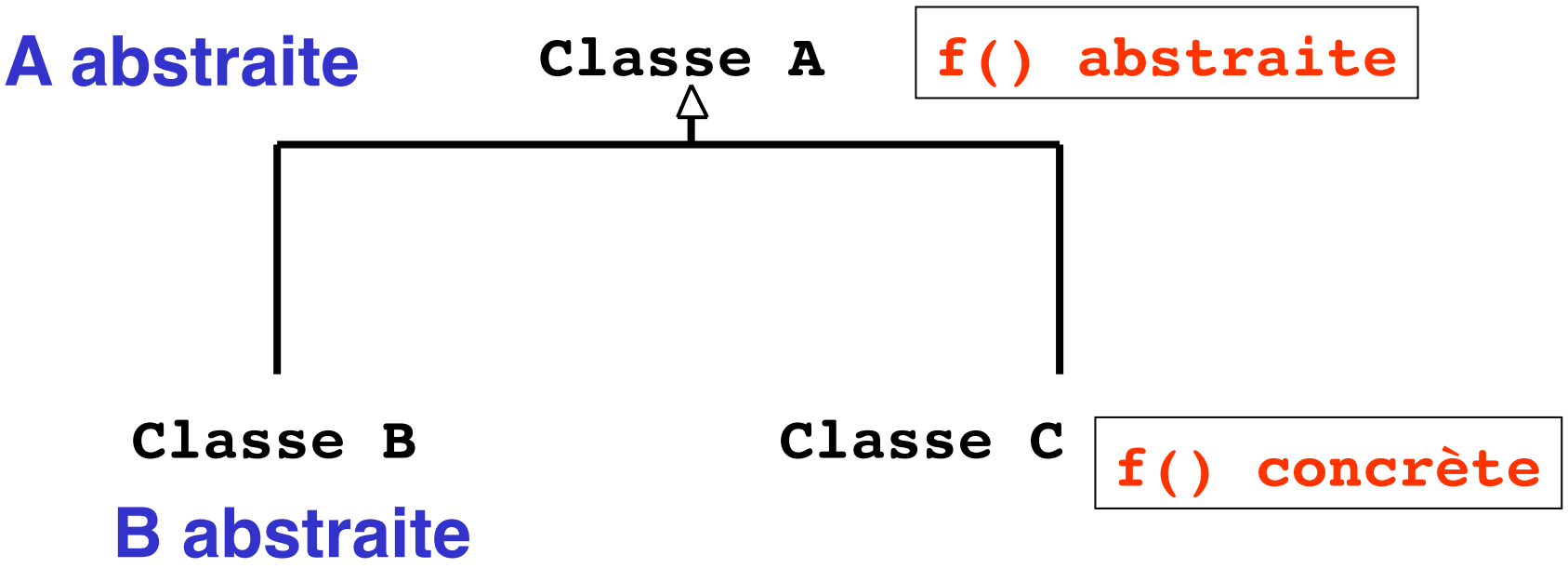
Règles

- **Toute classe possédant une méthode abstraite est abstraite**
- **Une classe peut être abstraite même si elle n'a pas de méthode abstraite**

"Toute classe qui possède une méthode abstraite est abstraite"

définit

hérite de



Toujours dans la classe Produit …

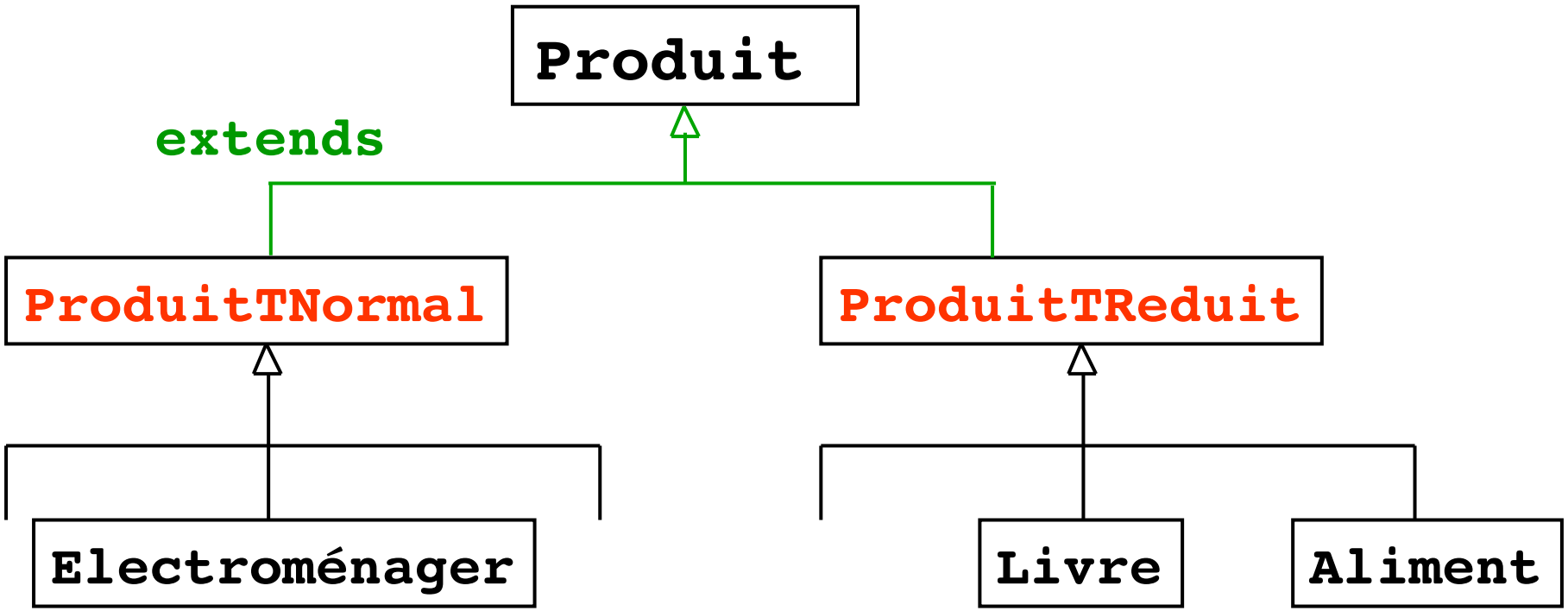
```
public String infos()
{
String s = reference + ' ' + designation;
s += '\n' + "prix HT: " + prixHT
s += '\n' + "prix TTC: " + lePrixTTC();
return s;
}
```

lePrixTTC() est abstraite

N'est-ce pas gênant ?

Non : on regarde plus loin …

Déclarer la relation d'héritage



La classe ProduitTNormal

- **Rôle : concrétiser la méthode lePrixTTC(), version simple***

*prix TTC = prix HT + (prixHT * 19,6%)*

```
public double lePrixTTC()
{return getPrixHT() * 1.196;}
```

- **Faut-il des constructeurs ?**

on y répondra en regardant la branche d'héritage parallèle

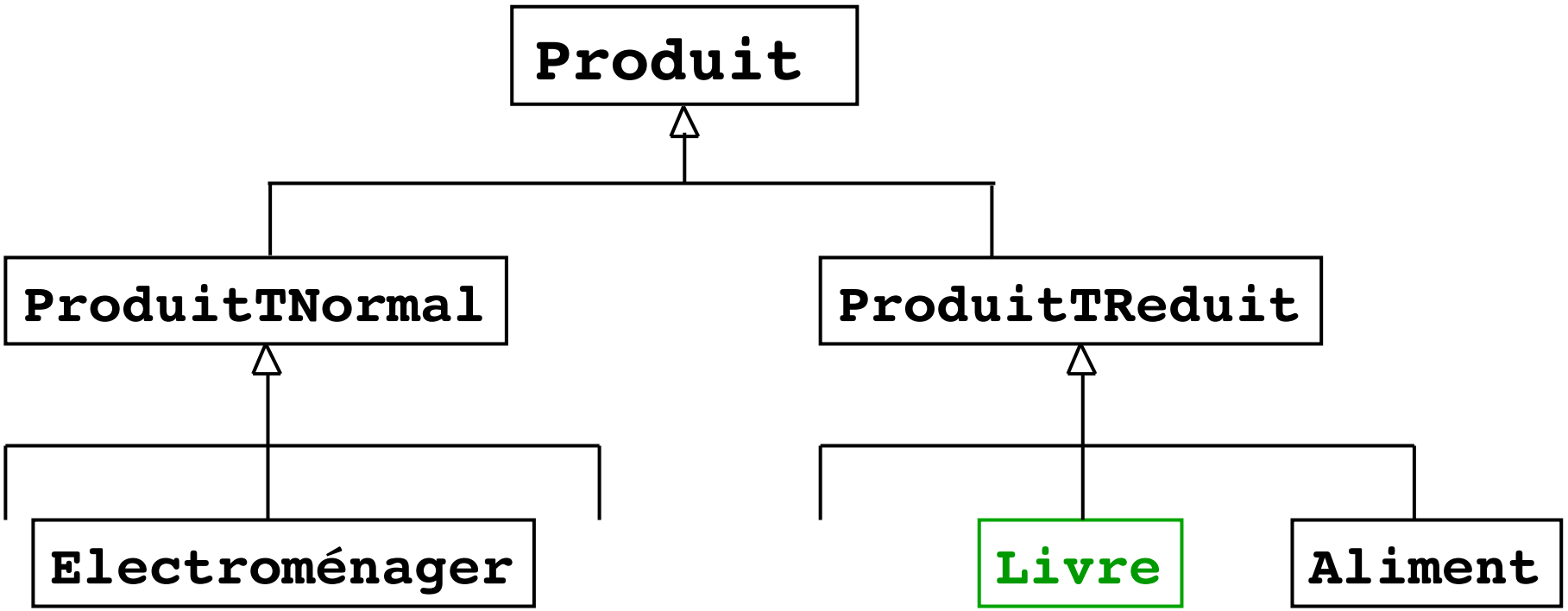
***Une version plus générale utiliserait une constante `TauxNormalTVA (exercice)`**

On récapitule :

- La déclaration d'héritage : extends
- La définition concrète de la méthode
- L'accès aux attributs grâce aux accesseurs

```
public class ProduitTNormal extends Produit
{


….

 public double lePrixTTC()
{return getPrixHT() * 1.196;}

}
```

Descendons dans l'autre branche …



La classe Livre

- **Une instance de Livre a 5 attributs**
  - **3 hérités de Produit (on ne les répète pas !)**
  - **éditeur, année**

```
public class Livre extends ProduitTReduit

{//attributs
private String editeur;
private int annee;

//accesseurs

public String getEditeur(){return editeur;}
public void setEditeur(String e){editeur=e;}

… idem pour annee

}
```

La classe Livre

- **Une instance de Livre a 5 attributs**
  3 hérités de Produit
  editeur, année

Règle générale :

- **chaque classe s'occupe des attributs qu'elle définit.**
- **pour les attributs hérités, elle délègue à ses super-classes**

Appliquer à :
constructeur
méthode infos()

La méthode infos() retourne :

le résultat de la méthode infos() héritée

+ présentation de editeur et année

```
public String infos()

{ return super.infos()
  + '\n' + editeur + ' ' +  annee;
}
```

super
= accès à la méthode *masquée*

Classe A

```
String f() {return "fA";}
String g() {return "gA";}
```


Classe B

```
String f() {return "fB " + super.f();}
String h() {return "hB " + super.f();}
String k() {return "kB " + super.g();}
```

Dans h() : super.f() bof - conception ?

Dans k() : super.g() NON - écrire g()

Règle : on utilise super.f() dans une nouvelle définition de f()

Classe A

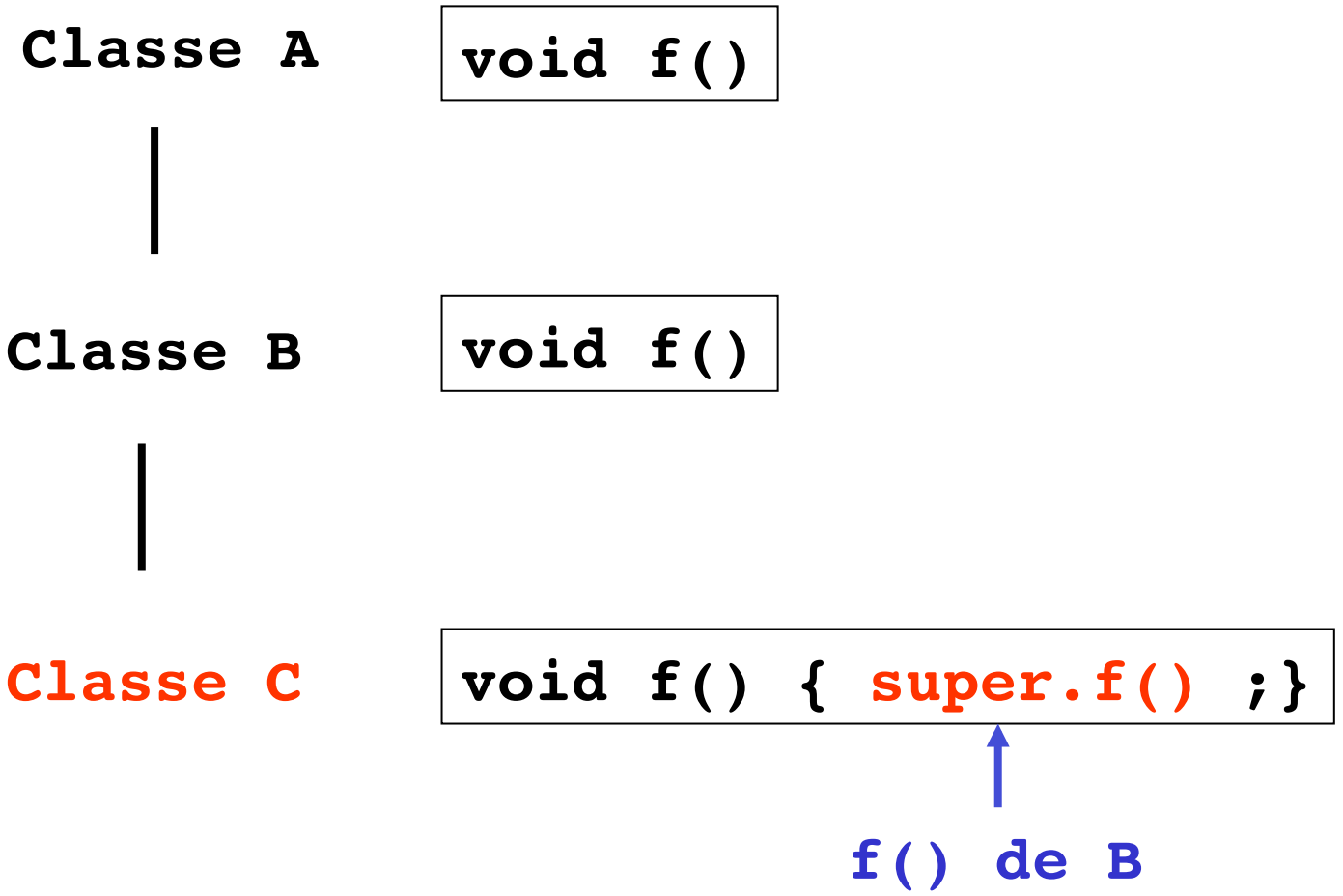
```
String f() {return "fA";}
String g() {return "gA";}
String i() {return "i "+f();}
```

Classe B

```
String f() {return "fB " + super.f();}
String h() {return "hB "+ super.f();}
String k() {return "kB " + g();}
```

```
B b = new B();
System.out.println(b.f()); fB fA
System.out.println(b.g()); gA
System.out.println(b.h()); hB fA
System.out.println(b.k()); kB gA
System.out.println(b.i()); i fB fA
```

On ne peut faire appel qu'à la méthode *masquée*



Aucun moyen d'appeler directement f() de A

Constructeur de Livre

Règle générale :

- **chaque classe s'occupe des attributs qu'elle définit**
- **pour les attributs hérités, elle délègue à ses super-classes**

Constructeur de Livre :

- délègue à ses super-classes l'initialisation des attributs hérités

- initialise editeur et an

Mais ...

dans un constructeur, on ne peut appeler qu'un constructeur de la super-classe directe

donc ProduitTReduit pour Livre

Il faut donc un constructeur dans ProduitTReduit

Qui ne sert que de "passeur"...

Produit

```
Produit(reference, designation, prixHT)
{initialise les attributs de même nom}
```

ProduitTR

```
ProduitTR (reference, designation,prixHT)
{ passe les paramètres à Produit }
```

Livre

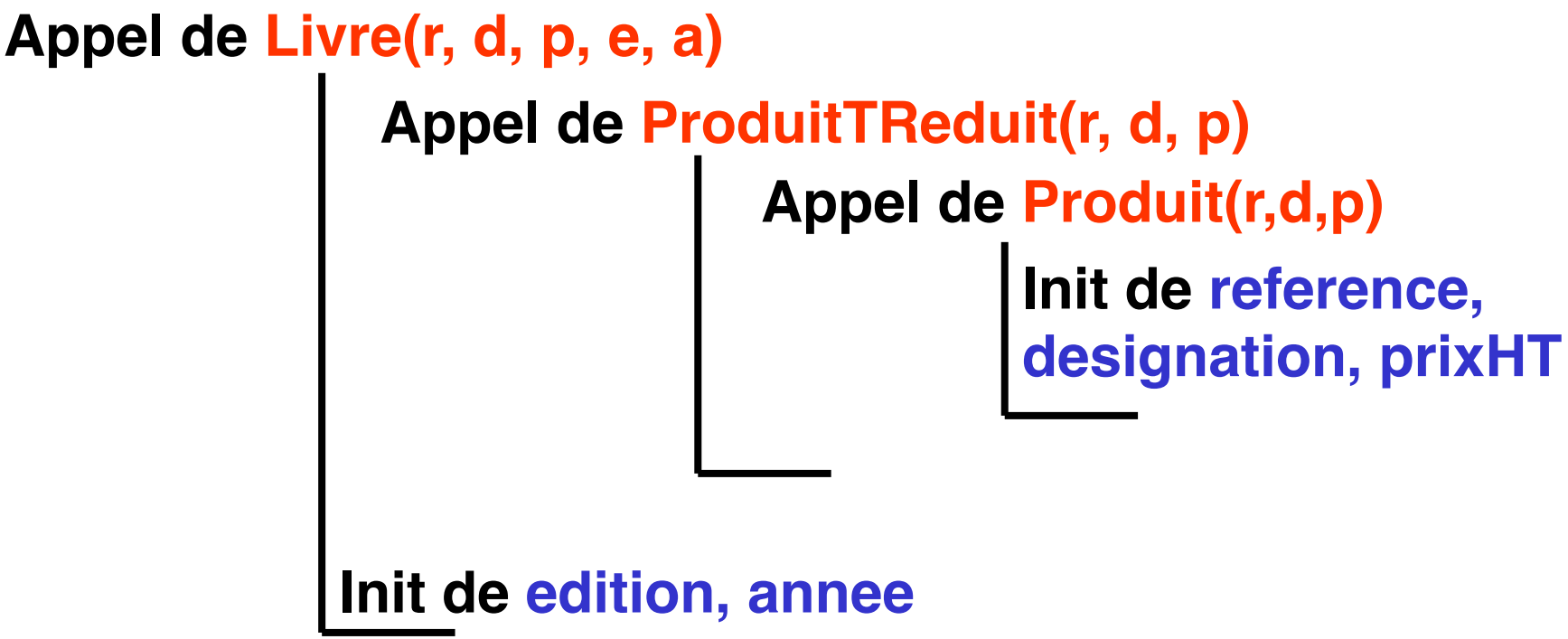
```
Livre (reference, designation, prixHT,
          editeur, an)
{ - passe les 3 premiers paramètres
  à ProduitTR
  - initialise les attributs editeur et annee }
```

```
public Livre(String reference,
             String designation, float prixHT,
             String editeur, int an)
{
     super(reference, designation, prixHT);
     this.editeur = editeur;
     this.annee = an;
}
```

```
public ProduitTReduit(String reference,
                      String designation,
                      float prixHT)
{
     super(reference, designation, prixHT);
}
```

Initialisation d'un Livre

```
Livre L = new Livre(r, d, p, e, a);
```



L'appel super(...) est la première instruction du constructeur.

L'exécution d'un constructeur commence toujours par un appel à un constructeur de la super-classe directe

(sauf pour Object qui n'a pas de super-classe)

Ce peut être implicite : au besoin le compilateur insère l'appel

```
super();

// appel au constructeur sans paramètre
// de la super-classe directe
```

Exemple : classe Produit

```
public Produit(String reference,
      String designation, float prixHT)
      {
            super();

            this.reference = reference;
            this.designation = designation;
            this.prixHT = prixHT;
      }
```

super() fait appel à quel constructeur? **Celui de Object**

Qui n'en définit pas... **Donc constructeur par défaut**