
3. Définition de classes

Documentation et exemple d'utilisation de la classe CompteS (up5.mi.pary.jc)

```
/** CompteS (Compte Simplifiée) est une classe d'initiation permettant
 * d'implémenter des comptes bancaires <br/>
 * en particulier, elle permet de<ul>
 * <li>Créer des comptes</li>
 * <li>Enregistrer des crédits et des débits</li>
 * <li>Consulter le solde et l'historique des opérations</li>
 * <li>Consulter et modifier le découvert autorisé</li>
 * <li>Savoir si le solde du compte est insuffisant</li>
 * </ul>
 */
```

Les fonctions de la classe CompteS

CompteS

- + **CompteS(String titulaire,double versementInitial,double decouvertAutorise)**
- + **CompteS(String titulaire)**
- + **String getNomTitulaire()**
- + **void addOperation(double montant)**
- + **double getSolde()**
- + **String getHistorique()**
- + **void setDecouvertAutorise(double decouvertAutorise)**
- + **double getDecouvertAutorise()**
- + **boolean isSoldeInsuffisant()**

Utilisation de la classe CompteS

```
CompteS c1 = new CompteS("Dupond",200,1000);// un compte pour Dupond  
CompteS c2 = new CompteS("Durand");// un compte pour Durand
```

```
c1.addOperation(100);//100 de plus pour Dupond  
c1.addOperation(300);//300 de plus pour Dupond  
c2.addOperation(-50); //50 Euros de moins pour Durand  
c1.addOperation(-30); // 30 Euros de moins pour Dupond
```

```
System.out.println("Solde de Dupond:"+c1.getSolde());  
System.out.println(c1.getHistorique()+c2.getHistorique());  
System.out.print(« Le solde du compte de " + c1.getNomTitulaire( )+" est ");  
if (c1.isSoldeInsuffisant( ))  
    System.out.println("insuffisant");  
else  
    System.out.println("suffisant");
```

Solde de Dupond:570

[200, 100, 300, -30][-50]

Le solde du compte de Dupond est suffisant

Documentation de la classe CompteS

```
/** construit un compte pour une personne
 * nommée 'nomTitulaire' sans versementInitial */
public CompteS(String nomTitulaire) {...}

/** construit un compte pour une personne
 * nommée 'nomTitulaire' et avec un solde initial de 'versementInitial' */
public CompteS (String nomTitulaire, double versementInitial, double decouvertAutorise) {...}

/** @return le nom du titulaire de ce compte */
public double getNomTitulaire( ) {...}

/** enregistre une opération de 'montant' Euros sur ce compte
 * crédite le compte si le montant est positif, débite sinon */
public void addOperation(double montant) {...}

/** @return le solde de ce compte */
public double getSolde( ) {...}

/** @return une chaîne illustrant l'historique des opérations sur ce compte */
public String getHistorique( ) {...}

/** teste si le solde est insuffisant */
public boolean isSoldeInsuffisant( ) {...}
```

DEFINITION DE CLASSES

**Nous savons définir des classes d'utilitaires
(i.e. dont tous les membres sont statiques)**

Nous savons utiliser des classes (Compte, String, Date, BigInteger ...)

Nous apprenons maintenant comment définir des (vraies) classes

Illustrée par la définition des classes

CompteS : une classe Compte simplifiée

Compte : compte avec découverts autorisés

CompteI et Individu : le titulaire est un Individu

DEFINITION DE CLASSES

Avant de définir une classe :

Définir les fonctionnalités souhaitées
Écrire la documentation de la classe

Pour définir une classe :

Choisir les attributs
Définir les fonctions membres
(constructeurs et méthodes)

choix des attributs d'instance

Le problème du choix des attributs d'instance est analogue à celui du choix des champs d'une structure (ou record) dans des langages non orientés objets.

Choisir les attributs d'instance de la classe Compte, c'est décider quelles données vont être mémorisées pour chaque compte.

Considérons que un compte doit mémoriser :

- le nom du titulaire (une String)
- le solde du compte (un double)
- l'historique des opérations (une liste)
- le découvert autorisé

"Dupond"
370
[100,300,-30]
100

on choisit de représenter l'historique des opérations par une liste dans laquelle les montants des opérations sont rangés sous forme de Double dans l'ordre chronologique. (à l'indice i est rangée la $i+1^{\text{ème}}$ opération)

Encapsulation des données

La classe CompteS
a 4 attributs d'instance:

```
public class CompteS {  
    private String nomTitulaire;  
    private double solde;  
    private List<Double> historique;  
    private double decouvertAutorise;  
  
    .....  
}
```

Les attributs sont privés, ils ne peuvent être modifiés QUE par les fonctions membres de la classe.

Cette encapsulation des données garantit que, la classe étant correctement définie, une instance de la classe ne peut pas se retrouver dans un état incohérent.

Par exemple, pour la classe CompteS, l'utilisateur de la classe ne peut pas modifier directement le solde sans mettre à jour l'historique.

Le solde ne peut être modifié que par l'appel de addOperation ce qui garantit la valeur du solde reste compatible avec l'historique.

Définition de constructeurs

Le rôle d'un constructeur est de construire un objet en initialisant ses attributs.

Le rôle des constructeurs de la classe CompteS est de construire un Compte en initialisant les attributs titulaire , solde , historique et decouvertAutorise

new CompteS("Dupond",300,100)

"Dupond"
300
[300]
100

Définition d'un constructeur de la classe CompteS

Dans un constructeur,
this référence l'objet en cours de création

this.nomTitulaire désigne le titulaire du compte en création

this.solde désigne le solde du compte en création

this.historique désigne le vecteur de l'historique des opérations du compte en création

this.decouvertAutorise désigne le découvert autorisé du compte en création

/** construit un compte pour une personne

* nommée 'nomTitulaire', avec un versement initial et un découvert autorisé spécifique */

```
public CompteS(String nomTitulaire, double versementInitial, double decouvertAutorise){  
    this.nomTitulaire = nomTitulaire;  
    this.solde = 0;  
    this.historique = new ArrayList<Double>();  
    this.addOperation(versementInitial);  
    this.setDecouvertAutorise(decouvertAutorise);  
}
```

initialisation
des attributs
de l'objet créé

Définition de méthodes d'instances

Une méthode d'instance se définit
de façon analogue
à une méthode statique.

La différence est qu'il y a un **paramètre formel implicite**
correspondant à l'**objet auquel est appliquée la méthode**.
Cet objet est référencé par **this**.

```
/** enregistre une opération de 'montant' Euros sur ce compte
 * crédite le compte si le montant est positif, débite sinon
 * @param montant le montant de l'opération */
public void addOperation(double montant) {
    if (montant != 0) {
        this.solde = this.solde + montant;
        this.historique.add(new Double(montant));
    }
}
```

Définition des méthodes de la classe CompteS (2)

```
/** @return le nom du titulaire de ce compte */  
public String getNomTitulaire( ) {  
    return this.nomTitulaire;  
}  
  
/** @return le solde de ce compte */  
public double getSolde( ) {  
    return this.solde;  
}  
  
/** @return le découvert autorisé de ce compte */  
public double getDecouvertAutorise( ) {  
    return this.decouvertAutorise;  
}  
  
/** modifie le découvert autorisé de ce compte */  
public void setDecouvertAutorise( double decouvertAutorise) {  
    this.decouvertAutorise=decouvertAutorise;  
}  
  
/** teste si le solde est insuffisant */  
public boolean isSoldeInsuffisant( ) {  
    return this.solde < 0 ;  
}
```

Définition des méthodes de la classe CompteS (3)

```
/** @return une chaîne illustrant l'historique des opérations sur ce compte */  
public String getHistorique( ) {  
    return this.historique.toString( );  
}
```

this.historique est la liste où sont notées les différentes opérations effectuées sur ce compte.

this.historique.toString() est la représentation sous forme de chaîne de caractères de cette liste

Définition de l'autre constructeur de la classe CompteS(1)

```
public CompteS(String nomTitulaire){  
  
    this.nomTitulaire=nomTitulaire;  
    this.solde = 0;  
    this.historique = new ArrayList<Double>();  
    this.decouvertAutorise = 0;  
}
```

**Cette façon de faire a l'inconvénient
de répéter des instructions déjà présentes
dans le premier constructeur**

**Ce qu'il faut,
c'est pouvoir utiliser l'autre constructeur**

Définition de l'autre constructeur de la classe CompteS(2)

Il est souvent intéressant et utile
d'appeler **un constructeur à l'intérieur d'un autre constructeur**
de la même classe :
une syntaxe spéciale utilisant l'identificateur "**this**" le permet

```
/** construit un compte pour une personne  
* nommée 'nomTitulaire' sans versementInitial */  
public CompteS(String nomTitulaire){
```

```
    //Cette instruction doit être la première du constructeur.
```

```
    // appel du premier constructeur CompteS(String,double,double)  
    this(nomTitulaire,0,0);
```

```
}
```


Définition de la classe CompteS : le texte complet(1/3)

```
package up5.mi.pary.jc.compte;
import java.util.List;
import java.util.ArrayList;
public class CompteS{
    private double solde;
    private String nomTitulaire;
    private List<Double> historique;
    /** construit un compte pour une personne
        * nommée 'nomTitulaire', avec un versement initial et un découvert autorisé spécifique */
    public CompteS(String nomTitulaire, double versementInitial, double decouvertAutorise){
        this.nomTitulaire = nomTitulaire;
        this.solde = 0;
        this.historique = new ArrayList<Double>();
        this.addOperation(versementInitial);
        this.setDecouvertAutorise(decouvertAutorise);
    }
    /** construit un compte pour une personne
        * nommée 'nomTitulaire' sans versementInitial ni découvert autorisé */
    public CompteS(String nomTitulaire){
        this(nomTitulaire,0,0);
    }
}
```

Définition de la classe CompteS : le texte complet (2/3)

```
/** enregistre une opération de 'montant' Euros sur ce compte  
    * crédite le compte si le montant est positif, débite sinon*/
```

```
public void addOperation(double montant) {  
    if (montant !=0){  
        this.historique .add(new Double(montant));  
        this.solde = this.solde + montant;  
    }  
}
```

```
/** @return une chaîne illustrant l'historique des opérations sur ce compte */  
public String getHistorique( ) {  
    return this.historique.toString( );  
}
```

Définition de la classe CompteS : le texte complet (3/3)

```
/** @return le nom du titulaire de ce compte */  
public String getNomTitulaire( ) {  
    return this.nomTitulaire;  
}  
/** @return le solde de ce compte */  
public double getSolde( ) {  
    return this.solde;  
}  
/** @return le découvert autorisé de ce compte */  
public double getDecouvertAutorise( ) {  
    return this.decouvertAutorise;  
}  
/** modifie le découvert autorisé de ce compte */  
public void setDecouvertAutorise( double decouvertAutorise) {  
    this.decouvertAutorise=decouvertAutorise;  
}  
/** teste si le solde est insuffisant */  
public boolean isSoldeInsuffisant( ) {  
    return this.solde < 0 ;  
}
```

CONVENTIONS D'ECRITURE

Types d'identificateurs	convention	exemple
classe	le premier symbole est une majuscule	System Date Compte
variable, fonction, attribut	le premier symbole est une minuscule	solde jour
attribut constant	tout en majuscule	PI

Pour séparer les mots composant un identificateur, on met des majuscules.
exemple : readString StringBuffer nomTitulaire

this. implicite

A l'intérieur d'une fonction (constructeur ou méthode)
si aucun objet n'est indiqué pour un membre d'instance,
c'est de **this** qu'il s'agit.

```
public CompteS(String nom , double versementInitial, double decouvertAutorise){  
    nomTitulaire= nom; // équivalent à this.nomTitulaire=nom;  
    solde = 0; // équivalent à this.solde = 0;  
    historique = new ArrayList<Double>(); // équivalent à this.historique = new ArrayList<Double>  
    addOperation(versementInitial); //équivalent à this.addOperation(versementInitial  
    setDecouvertAutorise(decouvertAutorise);  
}
```

this.
est implicite pour
les membres d'instances

Toutefois, il est préférable de le mentionner explicitement

paramètres formels des constructeurs et attributs

Dans une fonction, si une **variable** et un **attribut** sont homonymes, **this** doit être mentionné explicitement pour faire référence à l'attribut

```
public CompteS(String nomTitulaire,double versementInitial,double decouvertAutorise){  
    this.nomTitulaire=nomTitulaire; // this. obligatoire  
    this.solde = 0; // this. non obligatoire  
    this.historique = new ArrayList<Double>( ); // this. non obligatoire  
    this.addOperation(versementInitial); // this. non obligatoire  
    this.setDecouvertAutorise(decouvertAutorise) ; // this. non obligatoire  
}
```

Il est habituel que les paramètres des constructeurs et des méthodes, s'ils correspondent à la valeur d'initialisation d'un attribut, aient le même nom que les attributs qu'ils initialisent.

La classe Individu

La classe Compte représente l'individu titulaire d'un compte par son nom représenté par une String.

D'autres renseignements sont aussi intéressants : prénom, date de naissance ...

Il faut alors définir une classe Individu

Documentation de la classe up5.mi.pary.jc.Individu

```
public class Individu {  
    /** crée un individu connaissant son 'nom', son 'prénom' et sa 'dateDeNaissance' */  
    public Individu(String nom,String prenom,Date dateDeNaissance) {...}  
  
    /** rend le nom de cet individu */  
    public String getNom( ) {...}  
  
    /** rend le prénom de cet individu */  
    public String getPrenom( ) {...}  
  
    /** rend la date de naissance de cet individu */  
    public Date getDateNaissance( ) {...}  
  
    /** rend une chaîne de caractères composée du nom et du prénom de cet individu */  
    public String toString( ) {...}  
  
    /** rend l'âge de cet individu */  
    public int getAge( ){...}  
}
```


Utilisation de la classe Individu

```
Date d1 = new GregorianCalendar(1989,Calendar.APRIL,6).getTime( );
```

```
Individu i1 = new Individu("Dupond","Sylvie",d1);
```

```
Date d2 = new GregorianCalendar(1989,Calendar.OCTOBER,6).getTime( );
```

```
Individu i2 = new Individu("Durand","Claire",d2);
```

```
System.out.println(new Date( ));
```

```
System.out.println(i1+" "+i1.getAge());
```

```
System.out.println(i2+" "+i2.getAge());
```

```
Mon Aug 18 11:51:17 CEST 2014  
Dupond Sylvie 25  
Durand Claire 24
```

La classe Individu

```
package up5.mi.pary.jc.compte;
```

```
import java.util.Date;  
import java.util.Calendar;  
import java.util.GregorianCalendar;
```

```
public class Individu {  
    /** le nom de cet individu */  
    private String nom;  
    /** le prénom de cet individu */  
    private String prenom;  
    /** la date de naissance de cet individu */  
    private Date dateNaissance;
```

La classe Individu

/** crée un individu connaissant son 'nom', son 'prénom' et sa 'dateDeNaissance'*/

```
public Individu(String nom,String prenom,Date dateDeNaissance) {  
    this.nom = nom;  
    this.prenom = prenom;  
    this.dateNaissance = dateDeNaissance;  
}
```

/** rend le nom de cet individu */

```
public String getNom( ){return this.nom;}
```

/** rend le prénom de cet individu */

```
public String getPrenom( ){return this.prenom;}
```

/** rend la date de naissance de cet individu */

```
public Date getDateNaissance( ){return this.dateNaissance;}
```

/** rend une chaîne de caractères composée du nom et du prénom de cet individu */

```
public String toString( ){return this.nom + " "+this.prenom;}
```

La classe Individu

*/** @return l'âge de cet individu */*

```
public int getAge( ) {
```

```
    GregorianCalendar dActuelle = new GregorianCalendar( );
```

```
    GregorianCalendar dNaissance = new GregorianCalendar();
```

```
    dNaissance.setTime(this.dateNaissance);
```

```
    int nbAnnee = dActuelle.get(Calendar.YEAR)-dNaissance.get(Calendar.YEAR);
```

```
    // retire une unité si on n'est pas encore arrivé à l'anniversaire
```

```
    dActuelle.set(Calendar.YEAR,dNaissance.get(Calendar.YEAR));
```

```
    if (dNaissance.after(dActuelle)) nbAnnee--;
```

```
    return nbAnnee;
```

```
}
```

La classe CompteI avec la classe Individu

CompteS : une classe Compte simplifiée

Compte : découverts autorisés

CompteI et Individu : le titulaire est un Individu

Avant
(avec Compte
et CompteS)

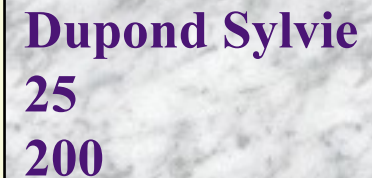
Le titulaire d'un compte est représenté par son nom
(une instance de la classe String).

Maintenant
(avec CompteI)

Le titulaire d'un compte est une instance de la classe Individu

La classe Comptel avec la classe Individu

```
Date dateNaissanceDupond =  
    new GregorianCalendar(1989,Calendar.APRIL,6).getTime( );  
  
Individu dupond = new Individu("Dupond","Sylvie", dateNaissanceDupond );  
  
Comptel compte = new Comptel(dupond);  
  
compte.addOperation(200);  
  
System.out.println(compte.getTitulaire( ));  
System.out.println(compte.getTitulaire( ).getAge( ));  
System.out.println(compte.getSolde( ));
```



Dupond Sylvie
25
200

La classe Comptel : changements par rapport à Compte(1)

L'attribut nomTitulaire est remplacé par titulaire

```
/** le titulaire de ce compte */  
private Individu titulaire;
```

```
public Comptel (Individu titulaire, double versementInitial, double decouvertAutorise) {  
    this.titulaire=titulaire;  
    this.solde = 0;  
    this.historique = new java.util.ArrayList<Double>();  
    this.addOperation(versementInitial);  
    this.decouvertAutorise=decouvertAutorise;  
}
```

```
public Comptel(Individu titulaire){  
    this(titulaire,0,0);  
}
```

La classe Comptel : changements par rapport à Compte (2)

La méthode getNomTitulaire est conservée mais sa définition est modifiée.

```
/** retourne le nom du titulaire de ce compte */  
public String getNomTitulaire( ) {  
    return this.titulaire.getNom( );  
}
```

Une nouvelle méthode rend le titulaire du compte.

```
/** retourne le titulaire de ce compte */  
public Individu getTitulaire( ) {  
    return this.titulaire;  
}
```


La classe Rationnel

OBJECTIFS

- pouvoir créer des rationnels connaissant le numérateur et le dénominateur
- pouvoir réaliser des additions et des multiplications
- pouvoir afficher des rationnels sous forme de fractions irréductibles

- création de rationnels
 - `Rationnel r1 = new Rationnel(5,12);`
 - `Rationnel r2 = new Rationnel(3,4);`
 - `Rationnel zero = new Rationnel(0);`
- additionner deux rationnels
 - `Rationnel s = r1.addition(r2);`
 - `Rationnel p = r1.multiplication(r2);`
- multiplier deux rationnels
 - `System.out.println(r1+" "+r2+"="+s);`
 - `System.out.println(r1+" "+r2+"="+p);`
- affichage de rationnels

L'interface de la classe "Rationnel"

/** crée un Rationnel de numérateur 'num' et de dénominateur 'den' */
public Rationnel(long num,long den){...}

/** crée un Rationnel égal à l'entier 'num'*/
public Rationnel(long num){...}

/** @return le rationnel somme de ce rationnel et de 'r'*/
public Rationnel addition(Rationnel r){...}

/** @return le rationnel produit de ce rationnel et de 'r' */
public Rationnel multiplication(Rationnel r){...}

/** @return une chaîne de caractères représentant ce rationnel avec
la notation habituelle sous forme de fractions irréductibles */
public String toString(){...}

Exemple d'utilisation simple de la classe Rationnel

```
package up5.mi.pary.jt.rationnel;  
import up5.mi.pary.jc.rationnel.Rationnel;  
  
public class TestRationnel1 {  
    public static void main(String [ ] args) {  
        Rationnel r1 = new Rationnel(5,12);  
        Rationnel r2 = new Rationnel(3,4);  
  
        System.out.println(r1+" "+r2+"="+"r1.addition(r2));  
        System.out.println(r1+"*"+r2+"="+"r1.multiplication(r2));  
  
    }  
}
```

$5/12 + 3/4 = 7/6$
 $5/12 * 3/4 = 5/16$

Somme de l'inverse des n premiers entiers

```
package up5.mi.pary.jt.rationnel;
import up5.mi.pary.jc.rationnel.Rationnel;import up5.mi.pary.term.Terminal;
public class TestRationnel2 {
/**@return le double somme de l'inverse des 'n' premiers entiers */
public static double sommeReelleDeInverseDesPremiersEntiers(int n){
double somme=0;
for (int i=1;i<=n;i++) somme=somme + 1d/i;
return(somme);
}
/**@return le Rationnel somme de l'inverse des 'n' premiers entiers */
public static Rationnel sommeRationnelleDeInverseDesPremiersEntiers(int n){
Rationnel somme=new Rationnel(0);
for (int i=1;i<=n;i++) somme=somme.addition(new Rationnel(1,i));
return(somme);
}
public static void main(String [] args){
Terminal term = new Terminal("calcul de sommes de rationnels",400,400);
int n = term.readInt("donner un entier ");
term.println("Somme : "+ sommeReelleDeInverseDesPremiersEntiers(n));
term.println("Somme : "+ sommeRationnelleDeInverseDesPremiersEntiers(n));}
```

donner un entier 10

Somme : 2.9289682539682538

Somme : 7381/2520

Définition de la classe Rationnel

La classe Rationnel définie ici a 2 attributs d'instance de type long:



Définition de la classe Rationnel

```
package up5.mi.pary.jc.rationnel;
import up5.mi.dupond.MathUtil; /* pour la fonction pgcd */

public class Rationnel {
    private long num,den;

    /** crée un Rationnel de numérateur 'n' et de dénominateur 'd' */
    public Rationnel(long num,long den){
        if (den==0) throw new ArithmeticException("dénominateur d'un rationnel nul");
        this.num=num;
        this.den=den;
        this.simplifier();
    }

    /** crée un rationnel égal à l'entier 'num' */
    public Rationnel(long num){
        // appel du constructeur Rationnel(long num,long den)
        this(num,1);
    }
}
```

Simplification et normalisation des rationnels

/normalise ce rationnel en le mettant sous forme
* irréductible avec un dénominateur positif */**

```
private void simplifier () {  
    if (this.num==0)  
        {this.den=1;}  
    else { long pgcd =  
        MathUtil.pgcd(Math.abs(this.num),Math.abs(this.den));  
        this.num=this.num/pgcd;  
        this.den=this.den/pgcd;  
        if (this.den<0) {  
            this.den= - this.den;  
            this.num= - this.num;  
        }  
    }}  
  
class Math{...  
public static long abs(long x){...}  
...}
```

Fonction membres privées

```
/**normalise ce rationnel en le mettant sous forme  
* irréductible avec un dénominateur positif */  
private void simplifier(){...}
```

simplifier est une fonction membre privée de la classe Rationnel.

**En effet, l'utilisateur de la classe Rationnel n'a pas à utiliser cette fonction.
(elle serait sans effet car les rationnels sont simplifiés dès leur construction)**

Si elle était public, elle apparaîtrait inutilement dans la documentation.

Définition des méthodes publiques la classe Rationnel

`/** @return le rationnel somme de ce rationnel et de 'r'*/`

public Rationnel addition(Rationnel r){

Rationnel res = new Rationnel(this.num*r.den+r.num*this.den,this.den*r.den);
return(res);
}

`/** @return le rationnel produit de ce rationnel et de 'r' */`

public Rationnel multiplication(Rationnel r){

Rationnel res = new Rationnel(this.num*r.num , this.den*r.den);
return(res);
}

`/** @return une chaîne de caractères représentant ce rationnel avec la notation habituelle sous forme de fractions irréductibles */`

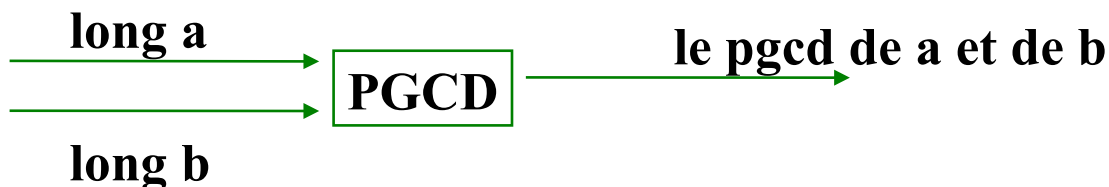
public String toString(){
return(this.num+"/"+this.den);
}

PGCD (Plus Grand Commun Diviseur)

Soit deux entiers strictement positifs a et b

Soit p le plus grand des nombres qui divisent à la fois a et b

p est le pgcd de a et b



$$42 = 2 * 3 * 7$$

$$\text{diviseurs}(42) = \{1, 2, 3, 6, 7, 14, 21, 42\}$$

$$28 = 2 * 2 * 7$$

$$\text{diviseurs}(28) = \{1, 2, 4, 7, 14, 28\}$$

$$544 = 2 * 2 * 2 * 2 * 2 * 17$$

$$\text{diviseurs}(544) = \{1, 2, 4, 8, 16, 17, 32, 34, 68, 136, 272, 544\}$$

$$119 = 7 * 17$$

$$\text{diviseurs}(119) = \{1, 7, 17, 119\}$$

Calcul du PGCD de deux entiers positifs

PGCD a b

544 119

(3) 425 119

(3) 306 119

(3) 187 119

(3) 68 119

(2) 68 51

(3) 17 51

(2) 17 34

(2) 17 17

(1) résultat = 17

Propriété

s

1. $\text{PGCD}(a, a) = a$
2. Si $a < b$ alors $\text{PGCD}(a, b) = \text{PGCD}(a, b - a)$
3. Si $a > b$ alors $\text{PGCD}(a, b) = \text{PGCD}(a - b, b)$

POUR CALCULER LE PGCD de a et b :
TANTQUE $a \neq b$ FAIRE
 SI $a < b$
 alors $b = b - a$
 sinon $a = a - b$

```
/** @return le pgcd de 'a' et de 'b'
 * 'a' et 'b' sont des entiers strictement positifs */
public static long pgcd(long a, long b){
    while (a!=b)
        if (a<b)
            b=b-a;
        else a=a-b;
    return(a);
}
```

Fonctions homonymes

Fonctions homonymes

- de classes différentes
- de même classe

Fonctions homonymes

1. fonctions homonymes de classe différentes

Des fonctions
de classes différentes

➡ "size" pour les classes ArrayList et HashMap
➡ "toString"

Il ne peut y avoir d'ambiguïté: l'objet sur lequel elles s'appliquent n'est pas de la même classe

```
java.util.Date date = new java.util.Date( );  
Object bigint = new java.math.BigInteger("12563820389");  
String s1 = date.toString( ); // toString( ) de java.util.Date  
String s2 = bigint.toString( ); // toString( ) de java.math.BigInteger
```

Fonctions homonymes

2. fonctions homonymes de même classe

Des fonctions homonymes de la même classe

Plusieurs fonctions d'une même classe peuvent porter le même nom à condition que le nombre ou le type des paramètres soit différents et permette ainsi au compilateur de les distinguer.



Erreur de compilation si seul le type de retour est différent

Fonctions homonymes

2. fonctions homonymes de même classe : exemple

Exemple : il y a 4 méthodes indexOf définies dans la classe String.

*/**returns the index within this string of the first occurrence of the specified character.**/*

```
public int indexOf(char ch){...}
```

*/**returns the index within this string of the first occurrence of the specified character,
* starting the search at the specified index. */*

```
public int indexOf(char ch, int fromIndex) {...}
```

*/**returns the index within this string of the first occurrence of the specified substring.*

```
public int indexOf(String str) {...}
```

*/**returns the index within this string of the first occurrence of the specified substring,
* starting at the specified index.**/*

```
public int indexOf(String str, int fromIndex){...}
```

Introduction à la définition de classes : définition de membres statiques

**Nous n'avons appris à définir
avec la classe CompteS que des membres d'instances.**

**Nous allons maintenant apprendre à définir,
avec la classe Compte, des membres statiques.**

Introduction à la définition de classes : définition de membres statiques

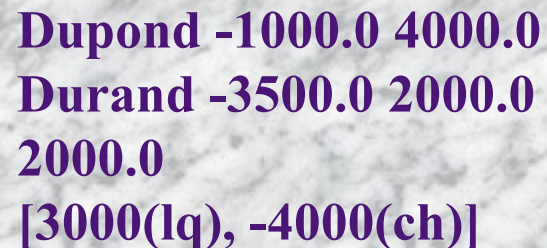
illustré par la définition de la classe Compte

la classe up5.mi.pary.jc.Compte :

```
/** classe d'initiation permettant d'implémenter
 * des comptes bancaires
 * en particulier, elle permet de
 * Enregistrer des crédits et des débits
 * Savoir si le compte a un solde suffisant
 * Consulter le solde et l'historique des opérations
 * Pouvoir consulter et modifier le découvert autorisé
 * Pouvoir consulter et modifier
 * le découvert autorisé par défaut
 * Pouvoir sauvegarder sur disque (non défini ici)
 */
```

Un programme utilisant la classe Compte

```
package up5.mi.pary.jt.compte;
import up5.mi.pary.jc.compte.Compte;
public class TestCompte1 {
    public static void main(String[] tArg) {
        Compte.setDecouvertAutoriseParDefaut(2000);
        Compte c1 = new Compte("Dupond",3000,4000);
        Compte c2 = new Compte("Durand");
        c1.addOperation(-4000,Compte.CHEQUE);
        c2.addOperation(-3500,Compte.LIQUIDE);
        System.out.println(c1.getNomTitulaire()+" "+c1.getSolde()+
            " "+c1.getDecouvertAutorise());
        System.out.println(c2.getNomTitulaire()+" "+c2.getSolde()
            +" "+c2.getDecouvertAutorise());
        System.out.println(Compte.getDecouvertAutoriseParDefaut(););
        System.out.println(c1.getHistorique( ););
    }
}
```



```
Dupond -1000.0 4000.0
Durand -3500.0 2000.0
2000.0
[3000(lq), -4000(ch)]
```

Les attributs d'instance de la classe "Compte"

Informations mémorisées pour chaque instance :

- le nom du titulaire,
- le solde du compte,
- l'historique des opérations
- le découvert autorisé

La classe Compte a 4 attributs d'instance:

```
private String nomTitulaire;  
private double solde;  
private List<String> historique;  
private double decouvertAutorise;
```

Les deux constructeurs de la classe Compte

```
/** construit un compte pour une personne  
 * nommée 'nomTitulaire' et avec un solde initial de 'versementInitial'  
 * et un montant de découvert autorisé égal à 'decouvertAutorise' */
```

```
public Compte( String nomTitulaire,  
              double versementInitial,  
              double decouvertAutorise){...}
```

```
/** construit un compte pour une personne  
 * nommée 'nomTitulaire' sans versement initial  
 * et un montant de découvert autorisé égal au montant de découvert autorisé par défaut  
 */
```

```
public Compte(String nomTitulaire){...}
```

Un attribut **de classe** de la classe "Compte"

Le montant du découvert autorisé par défaut à la construction d'un compte est un **donnée propre à la classe Compte : c'est donc un attribut de classe.**

//1 attribut privé de classe :

```
private static double decouvertAutoriseParDefaut = 0;
```

**Les attributs de classe sont repérés par le mot clé static.
(on dit aussi attributs statics)**

Définition du premier constructeur de la classe Compte

/** construit un compte pour une personne
* nommée 'nomTitulaire' et avec un solde initial de 'versementInitial'
* et un montant de découvert autorisé égal à 'decouvertAutorise' */

```
public Compte( String nomTitulaire,  
              double versementInitial,  
              double decouvertAutorise){  
  
    this.nomTitulaire=nomTitulaire;  
  
    this.solde = 0;  
  
    this.decouvertAutorise = 0;  
  
    this.historique = new ArrayList<String>( );  
  
    this.addOperation(versementInitial,Compte.LIQUIDE);  
    this.setDecouvertAutorise(decouvertAutorise);  
}
```

Définition de l'autre constructeur

```
/** construit un compte pour une personne
 * nommée 'nomTitulaire' sans versement initial
 * et un montant de découvert autorisé égal au montant de découvert autorisé par défaut
 */
public Compte(String nomTitulaire){
    this(nomTitulaire,0,Compte.decouvertAutoriseParDefaut);
}
```

Deux méthodes de classes pour consulter et modifier les découverts autorisés attribués lors de la création des comptes

\$<defCo>

```
/** @return le decouvert autorise attribué lors de la creation des comptes */  
public static double getDecouvertAutoriseParDefault( ) {  
    return Compte.decouvertAutoriseParDefault;  
}  
  
/** le decouvert autorise pour ce compte devient égal à 'decouvertAutorise'*/  
public static void setDecouvertAutoriseParDefault(double decouvertAutoriseParDefault) {  
    Compte.decouvertAutoriseParDefault = decouvertAutoriseParDefault;  
}
```

Il est naturel d'utiliser `decouvertAutoriseParDefault` comme nom de paramètre pour la méthode `setDecouvertAutoriseParDefault`.

Dans le corps de la fonction, la **donnée membre** se distingue du **paramètre** comme suit:

Compte.decouvertAutoriseParDefault : c'est la donnée membre
decouvertAutoriseParDefault : c'est le paramètre

Autres attributs **de classe** de la classe "Compte"

//4 attributs publiques de classe constants:

//Les quatre moyens de paiement

public **static** final int CB=0;

public **static** final int VIREMENT=1;

public **static** final int LIQUIDE=2;

public **static** final int CHEQUE=3;

//1 attribut privé de classe constant:

//Un tableau de chaînes pour l'affichage des moyens de paiement

private **static** final String [] tMoyenPaiement = {"cb","vi","lq","ch"};

(l'indice d'une chaîne dans tMoyenPaiement est égale à la constante correspondante:
ainsi : tMoyenPaiement[CB]=="cb" , tMoyenPaiement[VIREMENT]=="vi" ...)

Ajout d'une nouvelle opération sur un compte

```
/** enregistre une opération de 'montant' Euros
 * sur ce compte avec le 'moyenPaieement' indiqué */
public void addOperation(double montant,int moyenPaieement){
    if (montant !=0){
        this.solde = this.solde + montant;
        this.historique.add(this.getStringHisto(montant,moyenPaieement));
    }
}

/** retourne la chaîne représentant l'opération bancaire avec son montant
et le moyen de paiement montant de l'opération */
private static String getStringHisto(double montant,int moyenPaieement){
return montant + "("+tMoyenPaieement[moyenPaieement]+")";
}
```

getStringHisto est une **fonction membre privée** de la classe Compte. En effet, l'utilisateur de la classe Compte n'a pas à utiliser cette fonction.

Si elle était public, elle apparaîtrait inutilement dans la documentation.

Toute fonction dont l'utilisation à l'extérieur de la classe serait inutile ou néfaste doit être déclarée privée.

Compléments sur les attributs

- initialisation des attributs d'instance
- initialisation des attributs de classe
- attributs constants

initialisation des attributs d'instances

au moment
de la déclaration

```
private String nom;  
private double solde=0;  
private List<String> historique = new ArrayList<String>();  
  
public Compte(String nom,double montant,double decouvert){  
    this.nom=nom;  
    this.addOperation(montant);}
```

dans le constructeur

```
private String nom;  
private double solde;  
private List<String> historique;  
public Compte(String nom,double montant,double decouvert){  
    this.solde=0;  
    this.historique= new ArrayList<String>();  
    this.nom=nom;  
    this.addOperation(montant); }
```

Les attributs d'instances dont la valeur d'initialisation ne dépend pas des paramètres du constructeurs peuvent être initialisés au moment de la déclaration.

initialisation des attributs de classes

avec la déclaration

```
private static String tMois[12]={"Janvier","Février","Mars","Avril","Mai"
```

grâce à une suite d'instruction précédé du mot-clé static

```
public class Test{
```

```
    private static int[ ] tFact;
```

```
    static {
```

```
        tFact=new int[15];
```

```
        tFact[0]=1;
```

```
        for (int i=1;i<tFact.length;i++) tFact[i]=tFact[i-1]*i;
```

```
    }
```

```
}
```

Les bloc "static" sont exécutés une fois au moment du chargement de la classe.

S'il y en a plusieurs dans une même classe, ils sont exécutés en séquence dans l'ordre où ils apparaissent dans la définition.

Attributs constants

Ils sont déclarés avec le mot clé **final**

Exemple : un numéro de compte pour le client

```
private final int numeroDeCompte;
```

Les attributs final et static sont des constantes globales:

exemple: Math.PI

```
public class Math{  
    public static final PI=3.1415926535
```

Les attributs d'instance constants sont initialisables uniquement

- au moment de la déclaration
- ou dans le constructeur

Les attributs de classe constants sont initialisables uniquement

- au moment de la déclaration
- ou dans un bloc static

Visibilité

Visibilité

- des membres d'une classe
- des classes

Visibilité des membres d'une classe

L'INTERFACE D'UNE CLASSE EST
L'ENSEMBLE DES SIGNATURES (COMMENTEES)
DE SES MEMBRES PUBLICS

visibilités -->

PUBLIC

PROTECTED

PACKAGE

PRIVATE

FONCTION
(constructeurs
et méthodes)

les fonctions qui
apparaissent
dans la
documentation

les fonctions
à usage interne
de la classe

ATTRIBUTS

rare
sauf pour
les constantes

les attributs
sont le plus
souvent privées

Les différentes visibilité des membres d'une classe

public

visible sans restriction

protected

visible de la classe, des classes du même paquetage et des classes dérivées

package

visible de la classe et des classes du même paquetage

private

visible uniquement à l'intérieur de la classe

Attention

package est la visibilité par défaut
pour obtenir cette visibilité, **il ne faut rien indiquer**

Visibilité private et public: exemples

```
public class Essai {  
    private int xpriv;  
    public int xpub;  
  
    ...}
```

```
/* code d'une méthode en dehors de la classe Essai */  
Essai e = new Essai();
```

```
e.xpub=9; /* autorisé car visibilité public */
```

```
e.xpriv=2; /* interdit car visibilité private */
```

Remarque : lorsqu'une classe C ne définit aucun constructeur, un constructeur par défaut est défini automatiquement par le compilateur et est équivalent à la définition suivante:

```
public C() {} /* pas de paramètre, corps vide */
```

Visibilité package et public: exemples (1)

```
package dupond.visibilite;  
public class Essai {  
    private int xpriv;  
    int xpack;  
    public int xpub;  
    ...}
```

```
/* code d'une méthode en dehors de la classe Essai  
    mais du même paquetage*/  
Essai e = new Essai();
```

```
e.xpub=9; /* autorisé car visibilité public */  
e.xpack = 12; /* autorisé car visibilité package  
              et on est à l'intérieur du même paquetage */  
e.xpriv=2; /* interdit car visibilité private */
```

Visibilité package et public: exemples (2)

```
package dupond.visibilite;  
public class Essai {  
    private int xpriv;  
    int xpack;  
    public int xpub;  
    ...}
```

```
/* code d'une méthode en dehors de la classe Essai  
   mais d'un paquetage différent*/  
Essai e = new Essai();
```

```
e.xpub=9; /* autorisé car visibilité public */  
e.xpack = 12; /* interdit car visibilité package  
   et on n'est pas à l'intérieur du même paquetage */  
e.xpriv=2; /* interdit car visibilité private */
```

Visibilité d'une classe : public ou package

public

visible sans restriction

package

visible de la classe et des classes du même paquetage



pas de visibilité private ou protected pour les classes
UNE SEULE CLASSE "PUBLIC" PAR FICHIER

Attention

comme pour les membres, package est la visibilité par défaut

pour obtenir cette visibilité, **il ne faut rien indiquer**

Définition de classes

EXEMPLES avec définition de deux classes

Compte, Operation

pouvoir préciser d'autres renseignements que le montant de l'opération

Une opération consiste en un montant, une date, un commentaire ...

**L'opération n'est plus réduite à un simple montant,
on doit définir une classe (Operation) pour implémenter les opérations**

Yannick.Parchemal@parisdescartes.fr

La classe Operation

```
class Operation{  
    public Operation(double montant,Date date,String commentaire){...}  
    public String toString(){...}  
}
```

**La classe Operation n'étant utilisée qu'à l'intérieur de la classe Compte elle peut être définie dans le fichier Compte.java.
Elle ne peut alors être déclarée public**

**Une classe publique est définie dans un fichier de même nom suffixé par .java
D'autres classes non publiques peuvent être définies dans ce même fichier : ces classes non publiques ne sont visibles que dans leur paquetage.**

Une fonction de test pour CompteOS

```
/**Titre : Test de la classe CompteOS du package up5.mi.pary.jt.compte  
*/
```

```
package up5.mi.pary.jt.compte;  
import up5.mi.pary.jt.compte.CompteOS;  
import up5.mi.pary.term.Terminal;  
import java.util.Date;
```

```
public class TestCompteOS1 {  
    public static void main(String[] args) {  
        Terminal term = new Terminal("Compte avec opération détaillée",400,400);  
        CompteOS c1= new CompteOS("Dupond",200);  
        CompteOS c2= new CompteOS("Durant",1000);  
        c1.addOperation(450,new Date(),"cadeau");  
        c2.addOperation(-700,new Date(),"achat de logiciels");  
        c2.addOperation(4000,new Date(),"salaire");  
        c2.addOperation(200,new Date(),"prime");  
        term.println(c1.getHistorique( ));  
        term.println(c2.getHistorique( ));  
    }  
}
```


La classe Opération (1)

```
/**
 * Titre : Les comptes bancaires avec opérations détaillées <p>
 */
package up5.mi.pary.jc.compte;
import java.util.Date;

/** implemente les opérations sur des comptes bancaires */
class Operation {

    /** définition de la constante de formatage des dates */
    private final static java.text.DateFormat dateformat =
        new java.text.SimpleDateFormat("dd/MM/yy HH':'mm");
    /** définition de la constante de formatage des nombres décimaux */
    private final static java.text.DecimalFormat decimalformat =
        new java.text.DecimalFormat("###00.00");
```

La classe Opération (2)

/ le montant de cette opération */**

private double montant;

/ la date de cette opération */**

private Date date;

/ le commentaire associé à cette opération*/**

private String commentaire;

/ création d'une opération à la date 'date' d'un montant de 'montant'
avec le 'commentaire' associé*/**

public Operation(double montant, Date date, String commentaire) {

this.montant=montant;

this.date=date;

this.commentaire=commentaire;

}

public String toString(){

return dateFormat.format(date)+" : "+decimalformat.format(montant)+" "+commentaire;

}

}

Compte : ajout d'opérations

/ ajoute une opération à la date 'date' d'un montant de 'montant' Euros sur ce compte
* avec un 'commentaire'
* @param montant le montant de l'opération
* @param date la date de l'opération
* @param commentaire le commentaire associé à l'opération
*/**

```
public void addOperation(double montant, Date date, String commentaire){  
    this.historique.add(new Operation(montant, date, commentaire));  
    this.solde = this.solde + montant;  
}
```

/ ajoute une opération à la date courante d'un montant de 'montant' Euros sur ce compte
* @param montant le montant de l'opération */**

```
public void addOperation(double montant){  
    this.addOperation(montant, new Date(), "");  
}
```

Compte : autres méthodes modifiées

/ @return une chaîne illustrant l'historique des opérations sur ce compte */**

```
public String getHistorique() {  
    String res="";  
    // il serait mieux d'utiliser java.lang.StringBuilder  
    for (int i=0;i<this.historique.size();i++)  
        res+=this.historique.get(i)+"\n";  
    return(res);  
}
```

```
public String toString(){  
    return("Compte n° "+this.numCompte+ " "+this.titulaire);  
}
```

Définition de classes dérivées

Définition

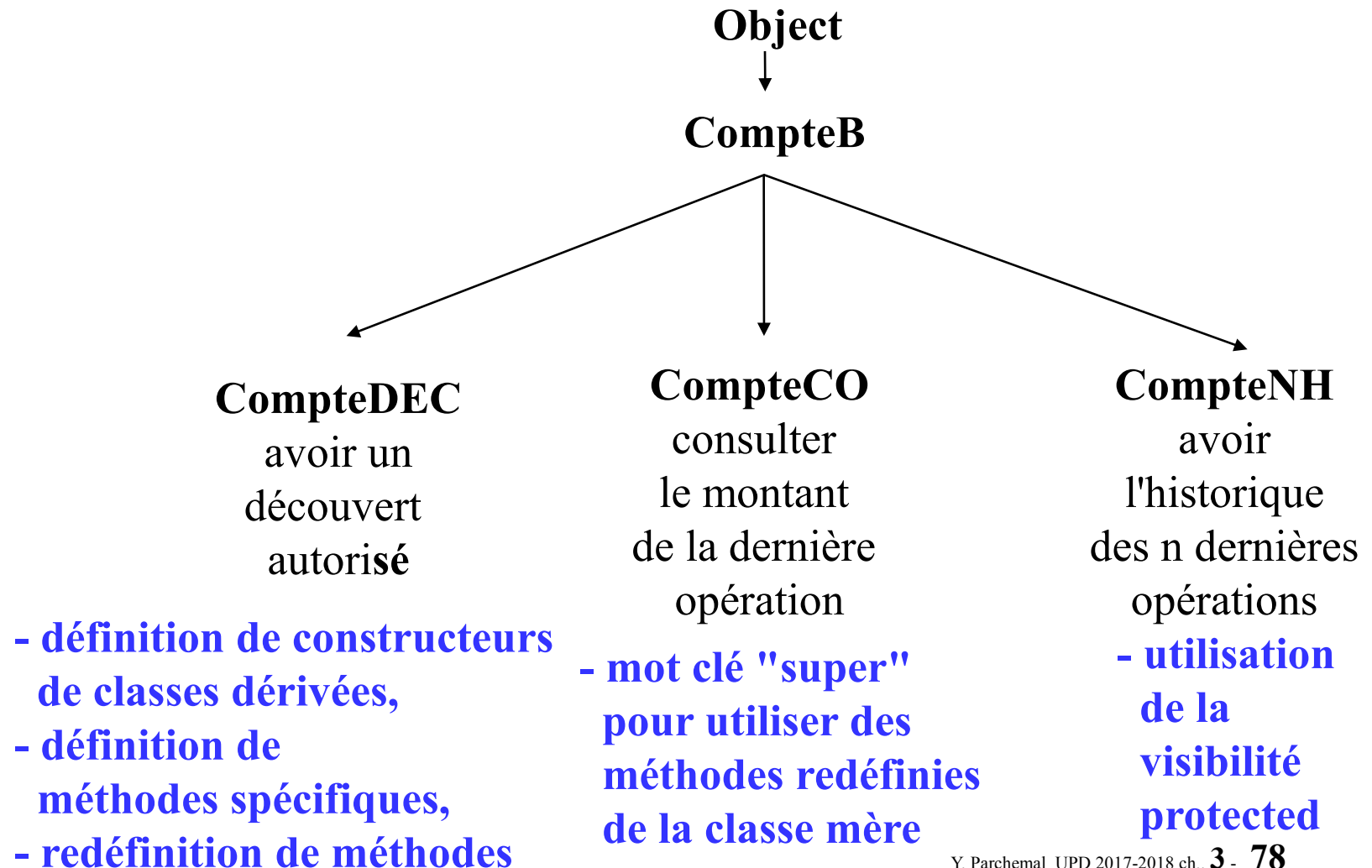
- Définition d'attributs spécifiques
- Définition des constructeurs
- Définition des méthodes spécifiques

Redéfinition (surcharge) de méthodes héritées

La définition de classes dérivées consiste à définir les membres spécifiques (attributs, constructeurs, méthodes) et à redéfinir éventuellement certaines méthodes héritées.

Classes dérivées de la classe CompteB

Pour illustrer l'étude des spécificités de la définition de classes dérivées, nous allons définir ces 3 classes dérivées de la classe CompteB



L'interface de la classe CompteS

Nous supposons la classe CompteS définie.

CompteB

```
+CompteB(String nom,double versementInitial)
+void addOperation(double montant)
+double getSolde( )
+String getHistorique( )
+boolean isSoldeInsuffisant( )
+String toString( )
```

CompteDEC : exemple d'utilisation

```
package up5.mi.pary.jt.compte;  
import up5.mi.pary.jc.compte.CompteDEC;
```

```
public class TestCompteDEC {
```

```
    public static void main(String[ ] args) {
```

```
        CompteDEC c = new CompteDEC("Dupond",200,1000);
```

```
        c.addOperation(-800);
```

```
        System.out.println("Solde insuffisant ? : "+c.isSoldeInsuffisant());
```

```
        System.out.println("Je dispose de "+c.getDebitPossible());
```

```
    }
```

```
}
```

CompteDEC

avoir un
découvert
autorisé

addOperation est une méthode héritée de **CompteB**

getDebitPossible est une méthode spécifique de la classe **CompteDEC**

isSoldeInsuffisant est une méthode redéfinie dans **CompteDEC**(définie dans **CompteB**)

Comptes avec découvert : l'interface

```
package up5.mi.pary.jc.comptederive;

public class CompteDEC extends CompteB {

    /** créé un compte pour 'nomTitulaire' avec un 'versementInitial' et un 'decouvertAutorise' */
    public CompteDEC(String nomTitulaire, double versementInitial, double decouvertAutorise) {...}

    /** rend le débit maximum possible pour rester avec un solde suffisant */
    public double getDebitPossible(){...}

    /** teste si le solde de ce compte est insuffisant */
    public boolean isSoldeInsuffisant(){...}

}
```

Définition de constructeurs de classes dérivées

la première instruction d'un constructeur est l'appel à un constructeur de la super classe.
L'appel au constructeur se fait grâce au mot clé 'super'

Cet appel permet l'initialisation des données membres héritées

Les instructions suivantes du constructeur
permettent d'initialiser les données membres spécifiques.

```
// Une donnée membre supplémentaire pour mémoriser le découvert autorisé  
private double decouvertAutorise;
```

```
/** créé un compte pour 'nomTitulaire' avec un 'versementInitial' et un 'decouvertAutorise'*/  
public CompteDEC(String nomTitulaire,double versementInitial,double decouvertAutorise) {  
    super(nomTitulaire,versementInitial); // appel au constructeur de la classe CompteS  
    this.decouvertAutorise=decouvertAutorise;  
}
```

Définition de méthodes spécifiques

```
/** rend le débit maximum possible pour rester avec un solde suffisant */  
public double getDebitPossible() {  
    double res= this.getSolde()+this.decouvertAutorise;  
    if (res>=0)  
        return res;  
    else return 0;  
}
```

**L'attribut d'instance solde est héritée de CompteB
mais n'est pas visible car il est privé.
C'est pourquoi on utilise ici getSolde()**

Redéfinition de méthodes

```
/** teste si le solde de ce compte est insuffisant */  
public boolean isSoldeInsuffisant( ){  
    return this.getSolde() < - this.decouvertAutorise;  
}
```

Une méthode redéfinie doit avoir exactement la même signature.

La classe compteDEC

```
package up5.mi.pary.jc.comptederive;

public class CompteDEC extends CompteB {
    private double decouvertAutorise;

    /** crée un compte pour 'nomTitulaire' avec un 'versementInitial' et un 'decouvertAutorise' */
    public CompteDEC(String nomTitulaire, double versementInitial, double decouvertAutorise) {
        super(nomTitulaire, versementInitial);
        this.decouvertAutorise = decouvertAutorise;
    }

    /** rend le débit maximum possible pour rester avec un solde suffisant */
    public double getDebitPossible() {
        double res = this.getSolde() + this.decouvertAutorise;
        if (res >= 0) return res; else return 0;
    }

    /** teste si le solde de ce compte est insuffisant */
    public boolean isSoldeInsuffisant() { return this.getSolde() < - this.decouvertAutorise; }
}
```

Comptes avec consultation de la dernière opération

CompteCO
consulter
le montant
de la dernière
opération

```
package up5.mi.pary.jc.comptederive;
```

```
public class CompteCO extends CompteB {
```

```
/** construit un compte pour 'nomTitulaire' avec un 'versementInitial' */  
public CompteCO(String nomTitulaire,double versementInitial) {...}
```

```
/** rend le montant de la dernière opération*/  
public double getDerniereOperation(){...}
```

```
/** ajoute une opération de 'montant' euros*/  
public void addOperation(double montant){...}  
}
```

Le constructeur

CompteCO
consulter
le montant
de la dernière
opération

```
package up5.mi.pary.jc.comptederive;
```

```
public class CompteCO extends CompteB {
```

```
    private double montantDerniereOperation=0;
```

```
    /** construit un compte pour 'nomTitulaire' avec un 'versementInitial' */
```

```
    public CompteCO(String nomTitulaire,double versementInitial) {  
        super(nomTitulaire,versementInitial);  
    }
```

Le constructeur d'une classe dérivée doit être défini même dans le cas où il se contente comme ici d'appeler le constructeur de la classe mère avec les mêmes paramètres.

Définition de méthodes spécifiques

CompteCO
consulter
le montant
de la dernière
opération

```
/** rend le montant de la dernière opération*/  
public double getDerniereOperation(){  
    return this.montantDerniereOperation;  
}
```


Utilisation de méthodes de la classe mère surchargées

```
/** ajoute une opération de 'montant' euros*/  
public void addOperation(double montant){  
    this.montantDerniereOperation=montant;  
    super.addOperation(montant);  
}  
}
```

CompteCO
consulter
le montant
de la dernière
opération

**Pour "forcer" l'appel à une méthode de la classe mère,
on utilise le mot clé **super****

La classe CompteCO

```
package up5.mi.pary.jc.comptederive;
```

```
public class CompteCO extends CompteB {  
    private double montantDerniereOperation=0;
```

```
/** construit un compte pour 'nomTitulaire' avec un 'versementInitial' */  
public CompteCO(String nomTitulaire,double versementInitial) {  
    super(nomTitulaire,versementInitial);  
}
```

```
/** rend le montant de la dernière opération*/  
public double getDerniereOperation(){  
    return this.montantDerniereOperation;  
}
```

```
/** ajoute une opération de 'montant' euros*/  
public void addOperation(double montant){  
    this.montantDerniereOperation=montant;  
    super.addOperation(montant);  
}  
}
```

CompteCO
consulter
le montant
de la dernière
opération

CompteNH

CompteNH
avoir
l'historique
des n dernières
opérations

```
package up5.mi.pary.jc.comptederive;
```

```
public class CompteNH extends CompteB {
```

```
    /** construit un compte pour 'nomTitulaire' avec un 'versementInitial' */  
    public CompteNH(String nomTitulaire, double versementInitial) {...}
```

```
    /** rend sous forme de String un historique des 'n' dernières opérations */  
    public String getDerniereOperation(int n){...}  
}
```

La classe CompteNH

CompteNH : avoir l'historique
des n dernières opérations

```
/** rend sous forme de String un historique des 'n' dernières opérations effectuées sur ce compte */  
public String getDerniereOperation(int n){  
    int nbOp= /* taille de l'historique */  
    if (nbOp<n) n=nbOp;  
    StringBuilder sb = new StringBuilder();  
    for (int i=0;i<n;i++)  
        sb.append(" "+/* élément de l'historique n°(nbOp-i-1) */ );  
    return sb.toString();  
}}
```

**Cette méthode nécessite d'avoir un accès plus fin
à l'historique des opérations.
La classe CompteB n'a pas été prévue pour cela.
Il va falloir la modifier.**

La classe Compte peut être modifiée de l'une des façons suivantes:

1- ajout de deux méthodes publiques donnant

- le nombre d'éléments de l'historique**
- un élément donné connaissant son indice**

```
public int getNbOperation( ) {return this.historique.size( );}  
public int getOperationNumero(int i) {return this.historique.get(i);}
```

2- ajout d'une méthode publique rendant une copie de l'historique

```
public List<Double> getVecteurHistorique( ){  
    return (List<Double>) this.historique.clone( );}
```

3- ajout d'une méthode protégée rendant l'historique

```
protected List<Double> getVecteurHistorique( ){return this.historique;}
```

4- visibilité protected pour la donnée membre "historique" de la classe mère

```
protected List<Double> historique;
```

La classe CompteNH

CompteNH : avoir l'historique
des n dernières opérations

1- ajout de deux méthodes publiques à la classe CompteB donnant

- le nombre d'éléments de l'historique
- un élément donné connaissant son indice

```
public int getNbOperation( ) {  
    return this.historique.size( );}  
public double getOperationNumero(int i) {  
    return this.historique.get(i).doubleValue( );  
}
```

/** rend un historique des 'n' dernières opérations effectuées sur ce compte*/

```
public String getDerniereOperation(int n){  
    int nbOp= this.getNbOperation();  
    if (nbOp<n) n=nbOp;  
    StringBuilder sb = new StringBuilder ();  
    for (int i=0;i<n;i++)  
        sb.append(" "+this.getOperationNumero(nbOp-i-1));  
    return sb.toString();  
}}
```

2- ajout d'une méthode publique rendant une copie de l'historique

```
public List<Double> getVecteurHistorique( ){  
    return (List<Double>) this.historique.clone( );  
}
```

3- ajout d'une méthode protégée rendant l'historique

```
protected List<Double> getVecteurHistorique( ){return this.historique;}
```

Choix 2 et 3

/** rend sous forme de String un historique des 'n' dernières opérations effectuées sur ce compte */

```
public String getDerniereOperation(int n){  
    List<Double> histo = this. getVecteurHistorique();  
    int nbOp= histo.size();  
    if (nbOp<n) n=nbOp;  
    StringBuilder sb = new StringBuilder ();  
    for (int i=0;i<n;i++)  
        sb.append(" "+ histo.get(nbOp-i-1));  
    return sb.toString();  
}}
```

4- visibilité **protected** pour la donnée membre "historique" de la classe mère
protected ArrayList historique;

Choix 4

/ rend sous forme de String un historique des 'n' dernières opérations effectuées sur ce compte*/**

```
public String getDerniereOperation(int n){  
    int nbOp= this.historique.size( );  
    if (nbOp<n) n=nbOp;  
    StringBuilder sb = new StringBuilder ();  
    for (int i=0;i<n;i++)  
        sb.append(" "+this.historique.get(nbOp-i-1) );  
    return sb.toString();  
}}
```


Quel choix effectuer ?

La classe CompteNH

CompteNH
avoir
l'historique
des n dernières
opérations

```
package up5.mi.pary.jc.comptederive;
```

```
public class CompteNH extends CompteB2 {
```

```
    /** construit un compte pour 'nomTitulaire' avec un 'versementInitial' */  
    public CompteNH(String nomTitulaire,double versementInitial) {  
        super(nomTitulaire,versementInitial);  
    }
```

```
    /** rend sous forme de String un historique des 'n' dernières opérations*/  
    public String getDerniereOperation(int n){  
        // une des versions proposées précédemment  
    }
```

Surcharge des données membres

PAS DE LIAISON DYNAMIQUE POUR LES ATTRIBUTS

L'attribut à utiliser
est déterminé à la compilation

```
class Test {  
    protected int x=8;  
}  
public class TestFinal extends Test{  
    private int x=9;  
    public static void main(String[] tArg){  
        TestFinal g = new TestFinal();  
        System.out.println(""+g.x+((Test)g).x);  
    }  
}
```

98



A éviter

DEFINITION DE CLASSES D'EXCEPTIONS

Définir ses propres classes d'exceptions

java.lang.Throwable

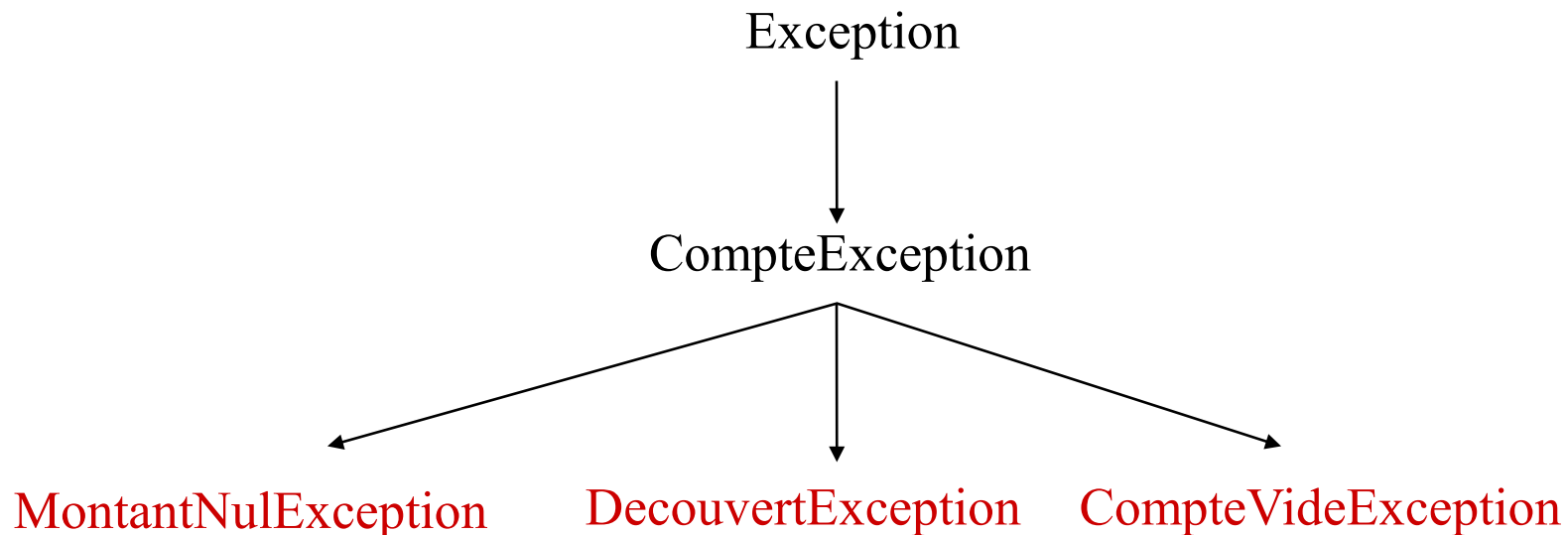
```
public class java.lang.Throwable extends java.lang.Object {  
    // Constructors  
    public Throwable( );  
    public Throwable(String message);  
  
    // Methods  
    .....  
    // rend un message détaillé  
    public String getMessage();  
    // affiche la pile d'exécution  
    public void printStackTrace();  
    //rend un message court  
    public String toString();  
}
```

java.lang.Exception

```
public class java.lang.Exception extends java.lang.Throwable
{
    // Constructors
    public Exception();
    public Exception(String message);
}
```

La plupart des classes d'exceptions ont l'allure de cette classe:
pas d'attributs ni de méthodes spécifiques
seulement des constructeurs

Créer ses propres classes d'exceptions



Les classes d'exceptions définies par l'utilisateur sont en général des sous-classes de Exception

Créer ses propres classes d'exceptions

```
public class CompteException extends Exception {  
    public CompteException(String s){super(s);}  
}
```

```
public class CompteVideException extends CompteException {  
    public CompteVideException(String s){  
        super("Aucune opération enregistrée sur le compte: "+s);}  
}
```

```
public class DecouvertException extends CompteException {  
    public DecouvertException(double montant,CompteE c){  
        super("Depassement du découvert autorise : operation impossible\n"+  
            "montant: "+montant+" solde:"+c.getSolde()+  
            " découvert autorise :"+c.getDecouvertAutorise());}  
}
```

```
public class MontantNulException extends CompteException {  
    public MontantNulException(String s){super(s);}  
}
```

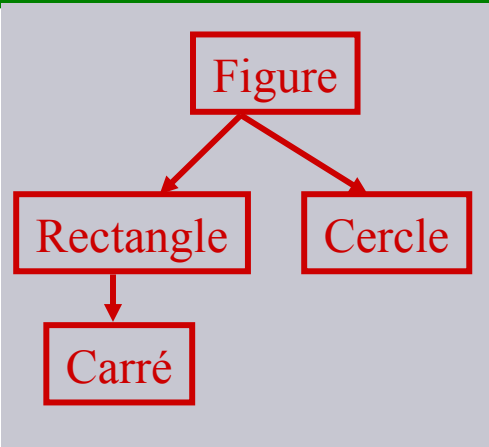

Lancer ses exceptions

```
/** enregistre une opération de 'montant' Euros sur ce compte avec le 'moyenPaiement' indiqué  
 * @param montant le montant de l'opération */  
public void addOperation(double montant) throws CompteException {  
    if (montant == 0)  
        throw new MontantNulException("Operation avec un montant nul !");  
    else if (montant + solde < -this.decouvertAutorise)  
        throw new DecouvertException(montant, this);  
    else {  
        this.historique.add(montant);  
        this.solde = this.solde + montant;  
    }  
}
```

Attraper ses exceptions

```
public class TestCompteE {
    public static void main(String[ ] args) {
        Terminal term = new Terminal("Compte avec gestion d'exception",300,300);
        CompteE c1 = new CompteE("Dupond",200,0);
        int choix;
        do { term.print("1:Credit \n2:Debit\n3:Solde\n4:Historique");
            do choix = term.readInt(""); while ((choix<0)|| (choix>4));
            try {
                switch (choix) {
                    case 1: c1.addOperation(term.readDouble("montant ? "));break;
                    case 2: c1.addOperation(-term.readDouble ("montant ? "));break;
                    case 3: term .println("Solde= "+c1.getSolde());break;
                    case 4: term .println("Historique: "+c1.getHistorique());break;}}
            catch (DecouvertException ex)
                {term .println("Operation ignoree : "
                    + "compte non suffisamment approvisionné"+ex);}
            catch (MontantNulException ex)
                {term .println(" Operation ignoree : entrer des nombres non nuls !");}
            catch (CompteException ex){ex.printStackTrace( ); }
        }
        while (choix!=0);
    }
}
```

Définition de classes abstraites



```
public class Figure {
```

```
...
```

```
public void afficher( ) {
```

```
    System.out.println("Je suis un "+this.getClass().getName( )  
        +" de perimetre "+this.getPerimetre( )+" et d'aire "+this.getAire( ));  
}
```

```
public static void main(String[ ] args) {  
    List<Figure> list = new  
    ArrayList<Figure>( );  
    list.add(new Rectangle(56,20));  
    list.add(new Cercle(30));  
    list.add(new Carre(30));  
    for (int i=0;i<list.size( );i++)  
        list.get(i).afficher();  
}
```

Je suis un Rectangle de perimetre 152 et d'aire 1120

Je suis un Cercle de perimetre 188.496 et d'aire 2827.43

Je suis un Carre de perimetre 120 et d'aire 900

Rectangle Carre et Cercle

```
public class Rectangle extends Figure{
    private double largeur,longueur;
    public Rectangle(double longueur,double largeur){
        this.longueur=longueur;this.largeur=largeur;}
    public double getPerimetre( ){return(2*(this.largeur+ this.longueur));}
    public double getAire( ){return(this.largeur* this.longueur);}
}

public class Carre extends Rectangle{
    public Carre(double longCote){
        super(longCote,longCote);}
}

public class Cercle extends Figure{
    private double rayon;
    public Cercle(double rayon){this.rayon=rayon;}
    public double getPerimetre( ){return(2*Math.PI*(this.rayon));}
    public double getAire( ){return(Math.PI* this.rayon*this.rayon);}
}
```

Classe abstraite : la classe Figure

Aucune instance directe de Figure ne sera créée.
getPerimetre et getAire sont définies dans les classes dérivées

Elles sont déclarées “**abstract**” dans la classe Figure

```
package up5.mi.pary.jc.abstrait;
```

```
public abstract class Figure {
```

```
public abstract double getPerimetre();
```

```
public abstract double getAire();
```

```
public void afficher( ){
```

```
    System.out.println("Je suis un "+getClass().getName()  
        +" de perimetre "+getPerimetre()+" et d'aire "+getAire());
```

```
}}
```

getPerimetre et getAire sont déclarées
dans la classe Figure sans être définies

Figure est une classe abstraite

Les énumérations

```
public enum Reponse {  
    OUI, NON, ANNULER;  
}
```

Correspond à :

```
public class ReponseOld {
```

```
    public final static ReponseOld OUI=new ReponseOld();  
    public final static ReponseOld NON=new ReponseOld();  
    public final static ReponseOld ANNULER = new ReponseOld();
```

```
    private ReponseOld(){}  
}
```

- Toutes les instances sont associées à une constante static
- Impossible de créer d'autres instances (constructeur privé)

Enum : exemple d'utilisation

```
public static void main(String[] args) {  
    Reponse rep=getReponse();  
    if (rep==Reponse.OUI)  
        System.out.println("la réponse est positive");  
    else if (rep==Reponse.NON)  
        System.out.println("la réponse est négative");  
}  
  
public static Reponse getReponse(){  
    ...  
}
```

Enum : utilisation du switch possible

```
public static void main(String[] args) {  
    Reponse rep=getReponse();  
    switch (rep){  
        case OUI : System.out.println("la réponse est positive");  
        case NON : System.out.println("la réponse est négative");  
        default:  
    }  
}
```

- Switch est utilisable avec les types énumérés
- Dans les case du switch, le nom de l'enum est implicite.

Enum : constructeurs avec paramètres

```
public enum Mois {
```

```
JANVIER(1),FEVRIER(2),MARS(3),AVRIL(4),MAI(5),JUN(6),JUILLET(7),AOUT(8)  
,SEPTEMBER(9),OCTOBRE(10),NOVEMBRE(11),DECEMBRE(12);
```

```
private int num;
```

```
private Mois(int num){  
    this.num=num;
```

```
}
```

```
public int getNum() {  
    return num;  
}
```

```
}
```

```
// Mois.JANVIER.getNum() vaut 1
```

Parenthèses facultatives

```
public enum Reponse {  
    OUI,NON,ANNULER;  
}
```

Ou

```
public enum Reponse {  
    OUI(),NON(),ANNULER();  
}
```

Dans le cas de constructeur sans paramètre, les parenthèses peuvent être omises.

Méthode applicable à tous les Enum

Exemple pour l'Enum Mois

public static Mois valueOf(String name)

Exemple : Mois mois = Mois.valueOf("FEVRIER");

public static Mois[] values()

Exemple : Mois[] tabMois= Mois.values();

public String name()

Exemple : System.out.println(Mois.JANVIER.name()); //JANVIER

public int ordinal()

Exemple : System.out.println(Mois.JANVIER.ordinal()); //0