Partie 2
Algorithmique orienté objet

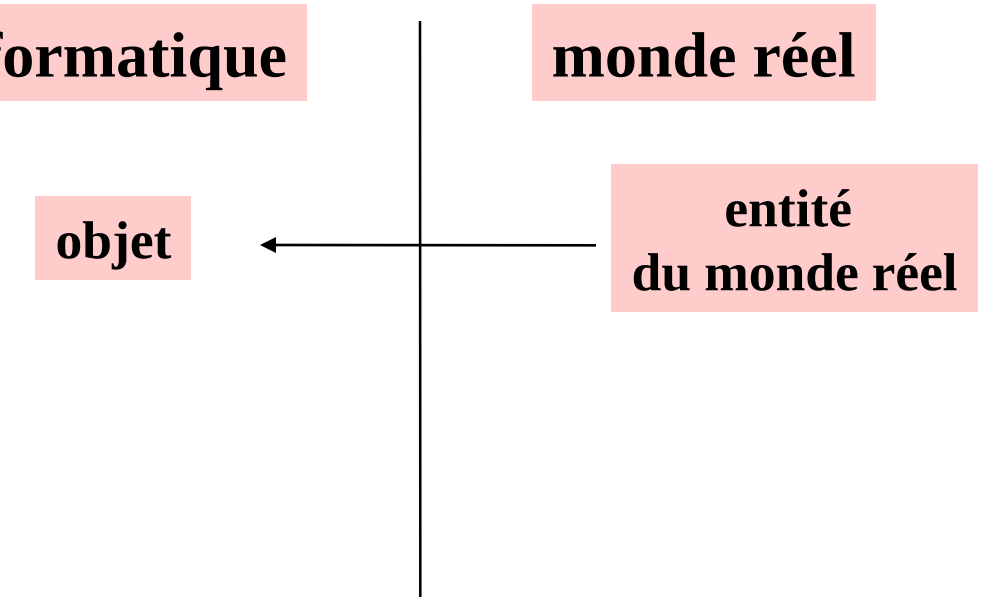
© Yannick.Parchemal@parisdescartes.fr

Les objets :première approche

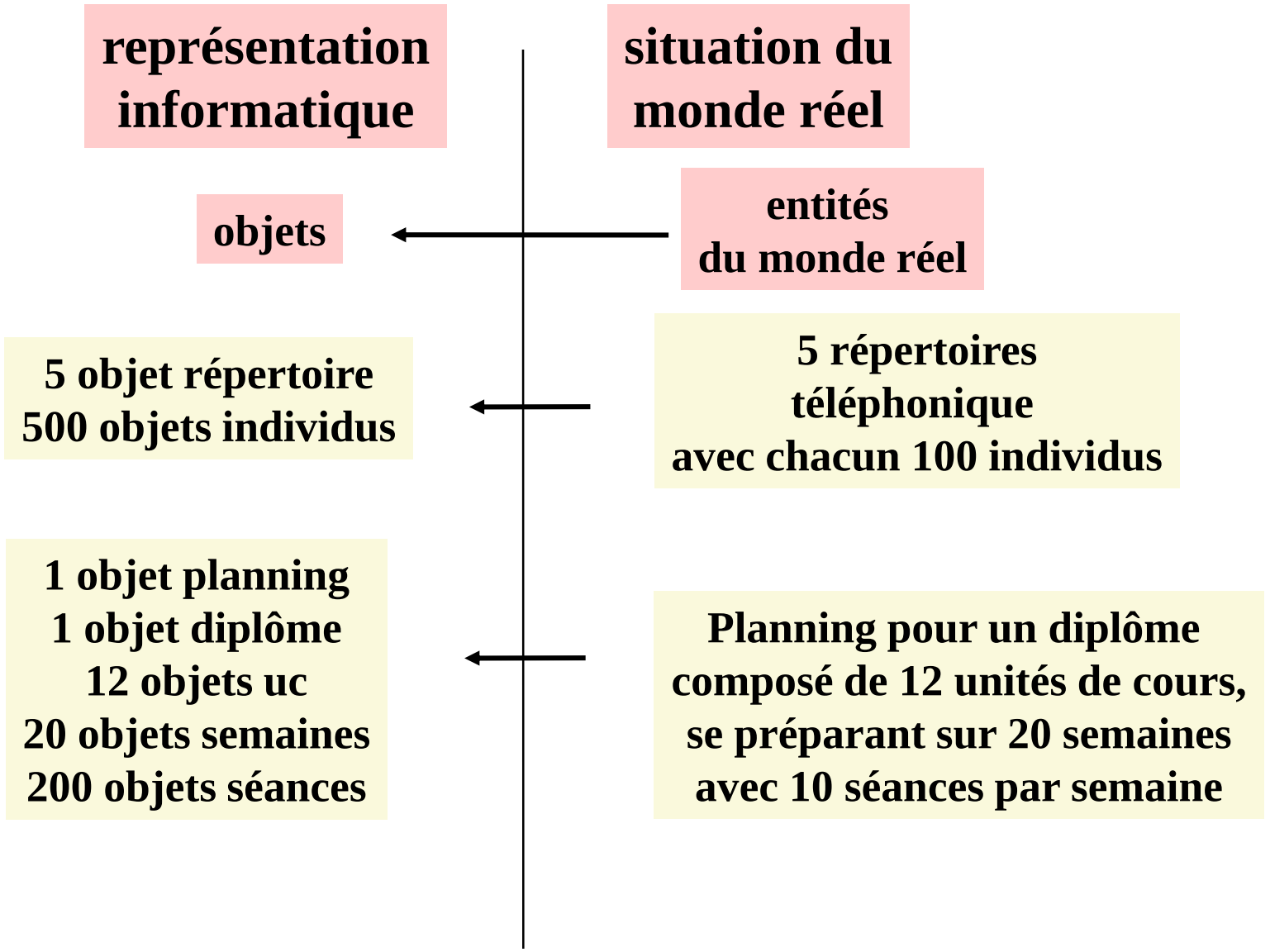
Un objet
est une entité informatique
créée et manipulée par le programme
correspondant à un entité concrète ou abstraite du monde réel

Exemple :

un répertoire téléphonique
un individu
une unité de cours
une figure géométrique
une question
un questionnaire

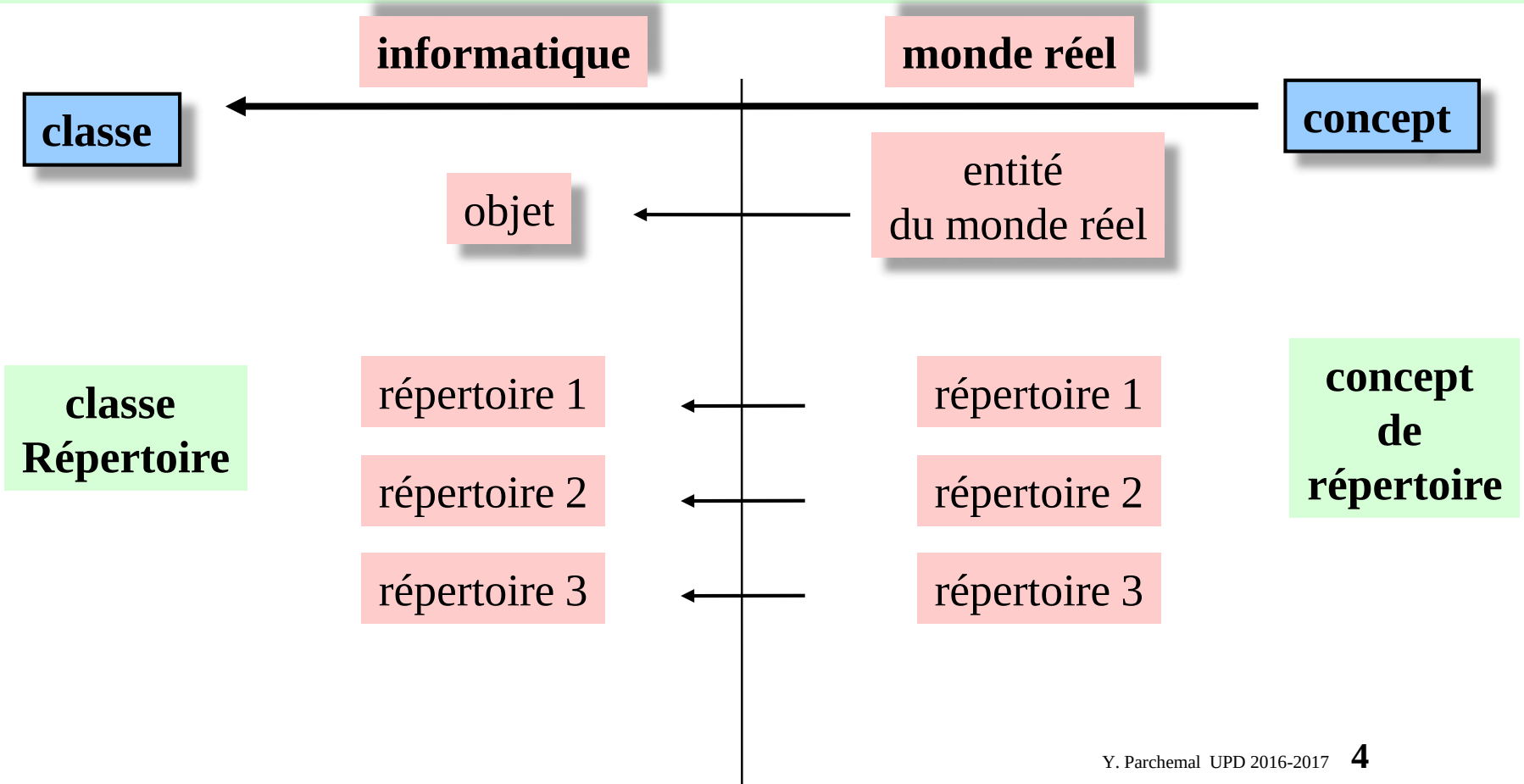


Correspondance objets-entités du monde réel : exemple



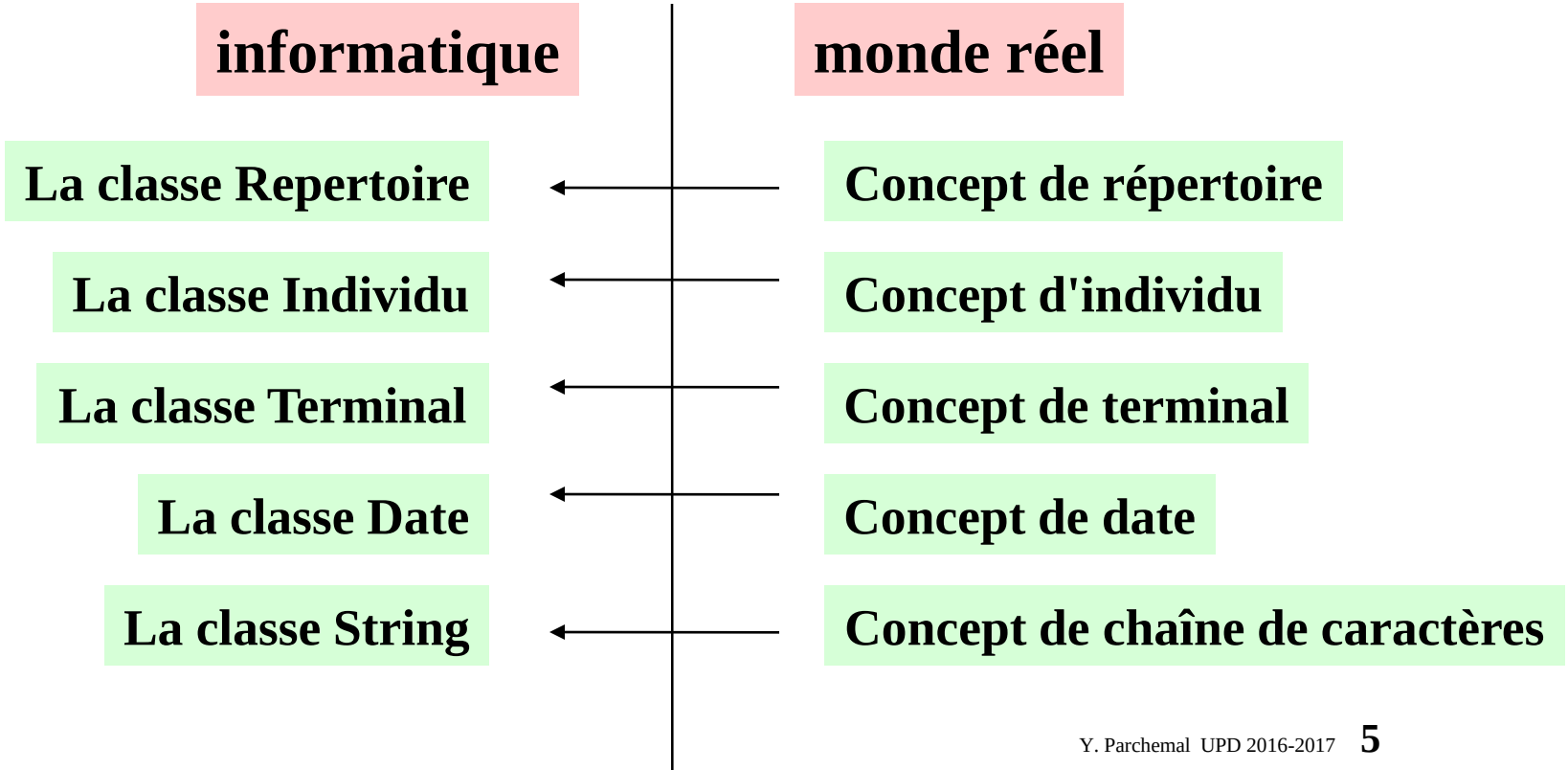
Classes

Certains objets possèdent de nombreux points communs car ils représentent des entités du monde réel d'une même catégorie, instances d'un même concept.
L'entité informatique correspondant à un concept du monde réel s'appelle une classe.



Concepts et classes

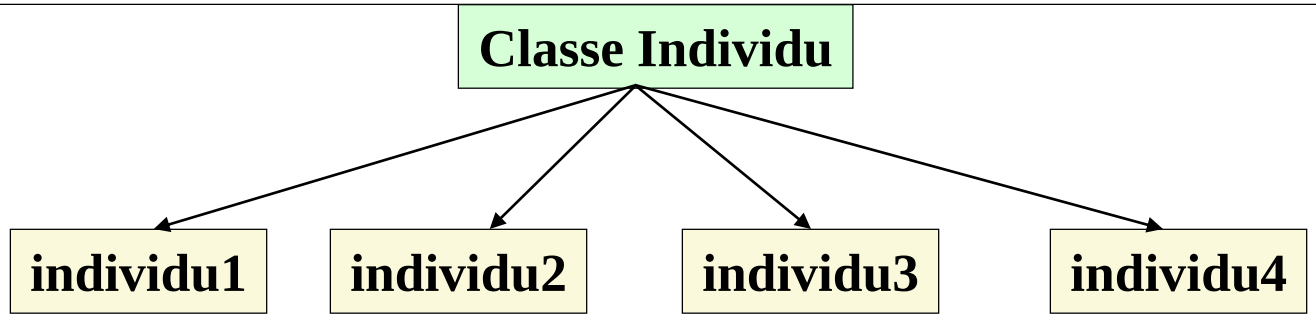
Une classe est une entité informatique correspondant à un concept concret ou abstrait du monde réel



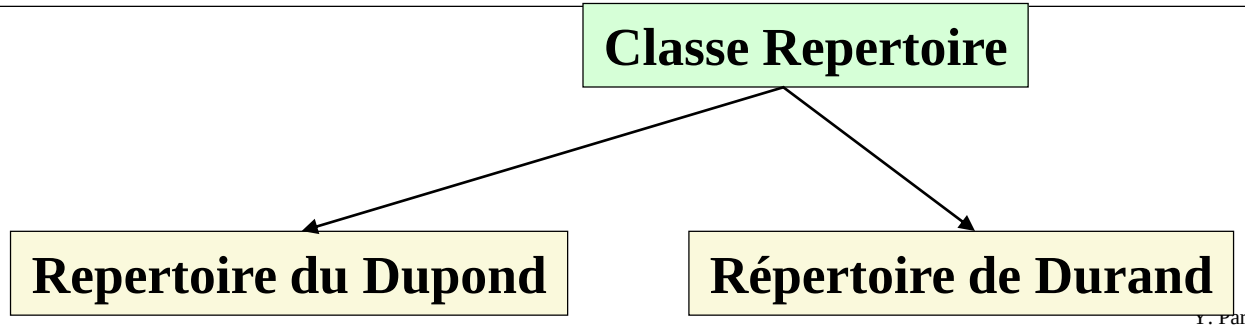
Un objet est une instance de sa classe

A chaque objet correspond une classe : la classe de l'objet
Un objet est une instance de sa classe.

Exemple 1 : la classe Individu
Chaque individu particulier est une instance de la classe Individu.



Exemple 2 : la classe Repertoire
Chaque répertoire particulier est une instance de la classe Repertoire.



Classes : constructeurs, attributs et méthodes

Une classe définit :

- des constructeurs pour créer des instances de la classe
- construction d'un répertoire connaissant le nom de son propriétaire

- les attributs permettant de définir l'état de chacune des instances
- le nom du propriétaire
- les éléments du répertoire

- les méthodes applicables à chacune de ses instances
- détermination d'un numéro de téléphone connaissant le nom d'une personne
- ajout d'un nouvel élément (nom-tel) dans le répertoire

objet = identité + état + comportement

chaque objet a :

- une identité qui lui est propre

même si une personne possède deux répertoires avec les mêmes entrées, ces deux répertoires sont bien distincts, ont une identité distincte.
Informatiquement parlant, l'identité est l'adresse de l'objet.

- un état : les informations qui lui sont propres
il est défini par des valeurs associées à chacun des attributs de la classe

pour un répertoire,
le nom de son propriétaire : Dupond
l'ensemble des personnes répertoriées : {Durand (01 44 56 23 24),…}

- un comportement défini par les méthodes qui lui sont applicables

pour un répertoire,
ajout d'un élément (nom-tel) dans le répertoire
obtenir un numéro de téléphone à partir d'un nom

classes et objets

classes **objets**

constructeurs —créent→ **identité**

attributs —pour définir→ **état**

méthodes d'instances —définissent→ **comportement**

Les constructeurs d'instances

La documentation des classes
permet de connaître les constructeurs disponibles
et donc de savoir comment créer des instances

Terminal
/ création d'un terminal connaissant son titre,** ** * sa largeur et sa hauteur initiales */** **+ Terminal(String titre,int largeur,int hauteur)**

```
// création d'un terminal de titre "essai" et de dimension 300*200
Terminal term = new Terminal("essai", 300 , 200);
// La valeur de term est maintenant une instance de la classe Terminal.
```

Les constructeurs sont des fonctions
- dont le nom est le nom simple de la classe
- sans type de retour

Construction d'instance : syntaxe

L'appel d'un constructeur est réalisé grâce à l'opérateur new .

Syntaxe de construction d'une instance:
new <nom_du_constructeur>(<paramètres>)

exemple : **new Terminal("essai", 300 , 200)**

Constructeur de la classe Individu

Individu
/ création d'un individu connaissant son nom, * son prénom et sa ville de naissance */ + Individu(String nom,String prenom,String ville)**

```
// création d'un individu nommé "Jean" "Dupond" né à Paris
Individu ind = new Individu("Dupond" , "Jean" , "Paris");
// La valeur de ind est maintenant une instance de la classe Individu.
```

Constructeurs de la classe CompteS

Une même classe peut avoir plusieurs constructeurs

CompteS

```
/** création d'un compte connaissant son titulaire,
   * avec un versementInitial
   * et le montant de decouvertAutorise*/
+ CompteS(String titulaire,double versementInitial,double decouvertAutorise)
/** création d'un compte sans versement initial
   * sans découvert autorisé spécifique
   * connaissant son titulaire*/
+ CompteS(String titulaire)
```

```
// création d'un compte bancaire pour Dupond
// avec un versement initial de 3000€ et un découvert autorisé de 2000€
CompteS c1 = new CompteS("Dupond",3000,2000);

// création d'un compte bancaire pour Dupond
CompteS c2 = new CompteS("Durand");
```

Autres exemples de constructions d'instances

```
// création d'un générateur de nombre aléatoire
Random gen = new Random( );

// création d'un objet Dimension de largeur 400 et de hauteur 200
Dimension d = new Dimension(400,200);
```

Remarque :
la classe String est un cas particulier : l'opérateur new n'est pas nécessaire pour créer une instance de cette classe
exemple:
String s = "bonjour";

Classes et méthodes d'instance

Une classe définit les méthodes applicables à une instance quelconque de cette classe

CompteS
/** rend le nom du titulaire de ce compte */   ***+ String getNomTitulaire( )***   ***/** rend le solde de ce compte */***   ***+ double getSolde( )***   ***/** ajouter une opération d'un montant donné sur ce compte*/***   ***+ void addOperation(double montant)***

```
// Application de addOperation à c avec comme paramètre -1000
// Ajout d'une opération de -1000€ sur le compte c
c.addOperation(- 1000);
```

Méthodes d'instance : ajouter une opération

Les méthodes d'instances sont des méthodes qui sont appliquées à une instance.

Syntaxe :
<instance>.<méthode>(<paramètres>)

```
c.addOperation(-1000)
```

applique
à l'instance c de la classe CompteS
la méthode addOperation de la classe CompteS
avec comme argument -1000

une méthode d'instance est toujours appliquée à une instance
la méthode et l'instance sont toujours de la même classe

Méthodes d'instance de la classe Terminal

Terminal
/** affiche un message dans ce terminal */ + void println(String message) /** rend un entier saisi à partir de ce terminal * après avoir affiché un message */ + int readInt(String message)

```
// exemple en Java:
// Application de la méthode println au terminal term
// avec en paramètre le message à afficher
// affiche sur le terminal term "Debut du tri"

term.println("Début du tri");
```

Autres exemples d'utilisation de méthodes d'instances

```
Individu ind = new Individu("Dupond" , "Jean" , "Paris");
// Application de la méthode getNom à l'individu ind
// ind.getNom( ) rend le nom de l'individu ind
String nom = ind.getNom( );

Random gen = new Random( );
// Application de la méthode nextInt au générateur gen
// gen.nextInt( ) rend un entier pseudo-aléatoire obtenu à partir du générateur gen
int x = gen.nextInt( );

Dimension dim = new Dimension(400,200);
// Application de getWidth à d
// Rend la largeur de d
int w = dim.getWidth( );
```

Utilisation de la classe CompteS

```
/** 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>
 * @author Yannick Parchemal
*/
```

Yannick.Parchemal@parisdescartes.fr

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
```

Fonctions utilisées dans l'exemple

```
CompteS c1 = new CompteS("Dupond",200,1000);
CompteS c2 = new CompteS("Durand");
c1.addOperation(100);
……
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");
```

CompteS

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

les 7 fonctions membres utilisées ici
- 2 constructeurs
- 5 méthodes d'instances

Les constructeurs de la classe Compte S

```
/** construit un compte pour une personne connaissant son nom,
   *  le montant du versement initial et le découvert autorisé
   * @param nomTitulaire        le nom du titulaire de ce compte
   * @param versementInitial   le solde initial de ce compte
   * @param decouvertAutorise      le découvert autorisé sur ce compte
*/
public CompteS(String nomTitulaire,
      double versementInitial,double decouvertAutorise){...}
```

exemple d'utilisation : CompteS c = new CompteS("Dupond",200,1000);

```
/** construit un compte pour une personne
   * connaissant son nom
   * sans découvert autorisé spécifique et sans versement initial
   * @param nomTitulaire        le nom du titulaire du compte */
public Compte(String nomTitulaire) {...}
```

exemple d'utilisation : CompteS c = new CompteS("Dupond");

Méthodes d'instance de la classe CompteS (1)

```
/** enregistre une opération sur ce compte
* @param montant le montant de l'opération en euros
*/
public void addOperation(double montant){...}
```

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

c.addOperation(300);
```

Méthodes d'instance de la classe CompteS (2)

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

```
CompteS c = new CompteS("Dupond",200,1000);
System.out.println(c.getNomTitulaire( ));
```

Dupond

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

```
System.out.println(c.getSolde( ));
```

200

```
/** @return vrai si le solde de ce compte est insuffisant faux sinon*/
public boolean isSoldeInsuffisant( ) {...}
```

```
c.addOperation(-20000);
System.out.println(c.isSoldeInsuffisant( ) ) ;
```

true

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

```
System.out.println(c.getHistorique( ));
```

[200 , -20000]

Méthodes d'instance de la classe CompteS (3)

```
/** @return le decouvert autorise  pour ce compte */
public double getDecouvertAutorise() {...}

/** modifie le decouvert autorise  pour ce compte*/
public void setDecouvertAutorise(double decouvertAutorise) {...}
```

```
Compte c = new CompteS("Dupond");
c.setDecouvertAutorise(3000);
System.out.println(c.getDecouvertAutorise( ));
```

3000

La classe java.lang.String

Elle implémente les chaînes de caractères

une syntaxe spécifique: les chaînes de caractères sont représentées par une suite de caractères entourée de guillemets

```
String s = "Bonjour le monde!";
```

un opérateur spécifique de concaténation : "+"

```
s = "Bonjour " + "le monde!";
```

il est impossible de modifier les caractères composant une String

Yannick.Parchemal@parisdescartes.fr

java.lang.String : deux méthodes d'instances

```
/**@return une copie de cette String  avec les minuscules converties en majuscules */
public String toUpperCase( ){...}

/** @return une String copie de cette String avec les majuscules converties en minuscules */
public String toLowerCase( ){...}
```

```
String s1="chaiNe 1    ";
String s2="cHAine 2";
System.out.println(s1.toUpperCase( ));
System.out.println(s2.toUpperCase( ));
System.out.println(s1.toLowerCase( ));
System.out.println(s1);
```

CHAINE 1
CHAINE 2
chaine 1
chaiNe 1

String : des chaînes de caractères non modifiables

```
/**@return une String égale à cette String dans laquelle toutes les occurrences d'un
   * caractères ont été remplacées par un autre caractère */
public String replace(char  oldChar, char  newChar){...}
```

```
String s1="chaiNe 1    ";
System.out.println(s1);
System.out.println(s1.replace(' ','*'));
System.out.println(s1);
```

chaiNe 1
chaiNe*1**
chaiNe 1

Les caractères composant une chaîne de caractères ne sont pas modifiables.
La méthode replace crée en fait une copie de la chaîne.

java.lang.String : autres méthodes d'instances

```
/**@return le nombre de caractères de cette String*/
public int length( ){...}
/**@return une String égale à cette String à laquelle on a supprimé les blancs et les caractères de code ascii <=  à 32 situés aux extremités*/
public String trim( ){...}

public char charAt(int  index){...}
/** @return une String égale à cette String à laquelle on a concaténé une autre chaine */
public String concat(String  str){...}
/**@return l'indice de la première occurrence d'une sous chaine dans cette String à partir d'un indice donné. Retourne -1 si aucune occurrence n'est trouvée
* @param str la sous-chaîne à chercher
* @param fromIndex l'indice à partir duquel commencer la recherche */
public int indexOf(String str,int fromIndex){...}
/**@return une copie de la partie de cette String entre un indice donné et la fin de cette String
* @param fromIndex l'indice à partir duquel la copie de cette chaîne est effectuée */
public String substring(int fromIndex){…}
```

java.lang.String : exemples d'utilisation

```
String s1="chaiNe 1    ";
String s2="cHAine 2";

System.out.println(s1.concat(s2));
System.out.println("%"+s1+"%"+s1.trim( )+"%");
System.out.println(s1.trim( ).concat(s2));
System.out.println(s1);
System.out.println(s1.replace(' ','*'));
System.out.println(s1.charAt(2));
```

```
chaiNe 1    cHAine 2
%chaiNe 1    %chaiNe 1%
chaiNe 1cHAine 2
chaiNe 1
chaiNe*1****
a
```

java.lang.String : méthodes d'instances

```
/**@return 0 si cette String est égale à une chaîne donnée, un entier négatif si elle lui est
   * inférieure lexicographiquement, positif sinon*/
public int compareTo(String  anotherString){...}
```

```
String s1="chaiNe 1    ";
String s2="cHAine 2";

System.out.println(s1.compareTo("autre chaine"));
System.out.println(s1.compareTo("encore une autre"));
System.out.println(s2.compareTo("cHAine 2"));
```

```
2
-2
0
```

La méthode public boolean equals(Object anObject)

```
/** Compares this string to the specified object. The result is true if and only if the
argument is not null and is a String object that represents the same sequence of
characters as this object.*/
public boolean equals(Object anObject){...}
```

```
String s1  = term.readString("Donner un nom");
String s2  = term.readString("Donner un 2eme nom");

System.out.println(s1==s2);
System.out.println(s1.equals(s2 ));
```

```
Donner un nom Dupond
Donner un 2eme nom Dupond

false
true
```

s1 et s2 ont pour valeurs deux chaînes différentes de même contenu.

== teste s'il s'agit du même objet (pas d'une copie)
equals teste l'égalité de contenu.

Exemple d'utilisation de la classe String (1)

```
/** retourne le nombre d'occurrence d'un caractère dans une chaîne*/
public static int getNbOcc(String s , char c){
   int cpt=0 ;
   for (int i=0;i<s.length( );i++)
         if (s.charAt(i) == c)
               cpt++;
   return cpt;
}
public static void main(String [] args){
System.out.println(getNbOcc("méthodes d'instance",'n'));
}
```

2

Exemple d'utilisation de la classe String

```
/** retourne le nombre d'occurrence d'une sous-chaîne dans une chaîne*/
public static int getNbOcc(String s,String sch){
   int cpt=0 ;   // compte le nombre de fois que l'on trouve la sous chaine
   int from=0; // recherche à partir de l'indice zéro
   int ind;  // indice de la sous-chaine suivante trouvée
   do {
     ind = s.indexOf(sch,from);
     if (ind != -1) {cpt++; from=ind+1;}
  while (ind != -1);
  return cpt;
}
public static void main(String [] args){
    System.out.println(getNbOcc("fgerterheryu","er"));
}
```

3

Exemple d'utilisation de la classe String

```
/** retourne une chaîne de taille donnée
   * dont tous les caractères sont égaux à un caractère donné*/
public static String getStringFromChar(char c, int n){
   String res="";
   for (int i=0;i<n;i++) res+=c;
   return res;
}

public static void main(String [] args){
   System.out.println(getStringFromChar('a',4));
}
```

aaaa

Autre exemple d'utilisation de la classe String

```
/** retourne une copie d'une chaine donnée où toutes les majuscules
sont transformées en minuscules et inversement */
public static String inverseMajMin(String s){
String res="";
for (int i=0;i<s.length();i++) {
    char c = s.charAt(i);
    if (Character.isLowerCase(c))
            c = Character.toUpperCase(c);
        else if (Character.isUpperCase(c))
            c = Character.toLowerCase(c);
    res+=c;
    }
return res;
}
```

isLowerCase, isUpperCase
toLowerCase, toUpperCase
sont des méthodes statiques de la classe Character

La classe java.util.Date

La classe Date implémente la notion de Date indépendamment de tout système de calendrier, des zones horaires etc… Ceci est particulièrement important pour l'internationalisation des programmes.

Une Date est définie par le nombre de milli-secondes écoulées depuis la date de référence qui est le 1er janvier 1970 0H GMT

Remarque : la date de référence peut être "traduite" pour tout calendrier et pour n'importe quelle zone horaire.
L'instant du 1er janvier 1970 OH GMT est le même pour tous, il est simplement exprimé de façon différente.

Yannick.Parchemal@parisdescartes.fr

La classe java.util.Date : un premier exemple

```
// Création d'une instance de la classe Date correspondant à la date actuelle
Date d1 = new Date( );
System.out.println("date actuelle : "+d1);

// Création d'une instance de la classe Date correspondant à la date de référence
Date d2 = new Date(0);
System.out.println("la date de référence : "+d2);

// Création d'une instance de la classe Date correspondant à 10^12 millisecondes
//  après la date de référence
Date d3 = new Date(1000000L *1000000);
System.out.println("1 billion de milli secondes apres le 1er janvier 1970 : "+d3);
```

```
date actuelle : Mon Aug 18 11:32:16 CEST 2014
la date de référence : Thu Jan 01 01:00:00 CET 1970
1 billion de milli secondes apres le 1er janvier
1970 : Sun Sep 09 03:46:40 CEST 2001
```

La classe java.util.Date : les membres publics

java.util.Date

+ Date( )
+ Date(long nbMilliDepuis010170)

+ boolean after(Date when)
+ boolean before(Date when)
+ boolean equals(Object obj)

+ long getTime( )
+ void setTime(long nbMilliDepuis010170)

+ String toString( )

La classe java.util.Date : les constructeurs

```
/** Allocates a Date object and initializes it so that it represents the time
at which it was allocated measured to the nearest millisecond. */
public Date( ){…}

/** Allocates a Date object and initializes it to represent the specified
number of milliseconds since January 1, 1970, 00:00:00 GMT.
@Param date  the milliseconds since January 1, 1970, 00:00:00 GMT */
public Date(long date){…}
```

Rappel : les constructeurs
- ont comme nom le nom de la classe
- n'ont pas de type de retour

La classe java.util.Date : les méthodes getTime et setTime

extraits de la documentation en ligne

Connaître le nombre de ms depuis le 1/1/70 d'une Date

```
/** Returns the number of milliseconds
   * since January 1, 1970, 00:00:00 GMT represented by this date. */
public long getTime( ) {…}
```

Modifier le nombre de ms depuis le 1/1/70 d'une Date

```
/** Sets this date to represent the specified number of milliseconds
   * since January 1, 1970 00:00:00 GMT.
   * @ param time the number of milliseconds. */
public void setTime(long time) {…}
```

utilisation des méthodes setTime et getTime

```
 public static void main(String[] args) {
Date date = new Date();
System.out.println("date de l'exécution :"+date);
long nbmilli = date.getTime( );
System.out.println("Depuis le 1/1/70 OH GMT: "+ nbmilli +" ms");
date.setTime(nbmilli +86400*1000); // 86400 sec par jour
System.out.println("24h plus tard:"+date);
System.out.println("Depuis le 1/1/70 OH GMT: "+date.getTime()+" ms");
}
```

```
date de l'exécution :Wed Aug 21 11:21:08 CEST 2013
Depuis le 1/1/70 OH GMT: 1377076868051 ms
24h plus tard:Thu Aug 22 11:21:08 CEST 2013
Depuis le 1/1/70 OH GMT: 1377163268051 ms
```

Les méthodes after et before

```
/** Tests if this date is after the specified date.
   * @param  when   a date
   * @return  true if this date is after the argument date; false otherwise. */
public boolean after(Date when){…}


/** Tests if this date is before the specified date.
 * @param when  a date
 * @return true if this date is before the argument date; false otherwise. */
public boolean before(Date when){…}
```

(extraits de Date.java)

Exemple d'utilisation de before et after

```
Date d1 = new Date( );
Thread.sleep(100); // dormir 100 millisecondes (environ)
Date d2 = new Date( );
System.out.print(d1.after(d2));
System.out.print (" "+d1.before(d2));
System.out.println(" "+ d1.equals(d2));
```

false true false

Les méthodes equals et toString de la classe Date

extraits de Date.java

```
/** Compares two dates. The result is true if and only if the argument is not null
   * and is a Date object that represents the same point in time, to the
   * millisecond, as this object.
   * Thus, two Date objects are equal if and only if the getTime method returns
   * the same long value for both.
   * @param obj  :  the object to compare with.
   * @return true if the objects are the same; false otherwise.
   * @see getTime */
public boolean equals(Object obj){…}

/** Creates a canonical string representation of the date.
   * The result is of the form "Sat Aug 12 02:30:00 PDT 1995".
   * @return  a string representation of this date.
public String toString() {…}
```

les méthodes equals et toString sont définies dans la classe Object

Différence entre la méthode equals et l'opérateur ==

```
Date d = new Date(0);
System.out.println(d==d);
System.out.println(d== dReference );
System.out.println(d.equals(dReference ));
```

```
true
false
true
```

== teste s'il s'agit du même objet (pas d'une copie)
equals teste l'égalité de contenu.

La méthode toString et println

La méthode toString est utilisée implicitement lors de
- **la concaténation de chaînes si l'une des opérandes est un objet**
- **l'affichage d'objets avec println (System.out.println)**

```
Date dReference = new Date(0);

System.out.println("d="+ dReference );
// équivalent à :
System.out.println("d="+ dReference.toString());
```

d= Thu Jan 01 01:00:00 CET 1970
d= Thu Jan 01 01:00:00 CET 1970

```
System.out.println(dReference);
// équivalent à :
System.out.println(dReference.toString());
```

Thu Jan 01 01:00:00 CET 1970
Thu Jan 01 01:00:00 CET 1970

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);
```

multiplier deux rationnels

```
Rationnel p = r1.multiplication(r2);
```

affichage de rationnels

```
System.out.println(r1+"+"+r2+"="+s);
System.out.println(r1+"*"+r2+"="+p)
```

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

Yannick.Par ;

L'interface de la classe "Rationnel"

```
/** crée un Rationnel de numérateur et dénominateur donné */
public Rationnel(long num,long den){…}

/** crée un Rationnel égal à  un entier donné*/
public Rationnel(long num){…}

/** @return le rationnel somme de ce rationnel et d'un rationnel donné*/
public Rationnel addition(Rationnel r){…}

/** @return le rationnel produit de ce rationnel et d'un rationnel donné*/
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 TestRationnel {
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

```
/**@return le double somme de l'inverse des 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 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

Somme des n éléments d'un tableau de Rationnel

```
/**@return la somme des éléments du tableau d'entiers 'tab' */
public static int sommeTableau(int [ ] tab){

int res=0;
for (int i = 0; i < tab.length ; i++)
    res += tab[i];
return(res);
}

/**@return la somme des éléments du tableau de rationnel 'tab' */
public static Rationnel sommeTableau(Rationnel [ ] tab){

Rationnel res=new Rationnel(0,1);
for (int i = 0; i < tab.length ; i++)
    res = res.addition(tab[i]);
return(res);
}
```

A propos des expressions avec new

new est un opérateur qui rend un objet.
Une expression composée avec new peut être utilisée comme n'importe quelle autre expression.

```
Rationnel r = new Rationnel(5,12);
Rationnel s = new Rationnel(3,4);

Rationnel res1 = r.addition(s);
Rationnel res2 = r.addition(new Rationnel(3,4));
Rationnel res3 = new Rationnel(5,12).addition(s);
Rationnel res4 = new Rationnel(5,12).addition(new Rationnel(3,4));
System.out.println(res1+"\n"+res2+"\n"+res3+"\n"+res4+"\n");
```

```
7/6
7/6
7/6
7/6
```

Membres d'une classe : les attributs d'instance

Les attributs d'instance sont des variables dont la valeur est propre à chaque instance

exemple : le nom d'un individu, la taille en pixels d'un terminal

Syntaxe : <instance>.<attribut>

exemple : la classe java.util.Dimension a deux attributs publics : width et height

```
Dimension dim = new Dimension(300,200);
System.out.println(dim.width);  // la largeur de l'objet dim de valeur 300
dim.width = 150; // la largeur de l'objet dim vaut maintenant 150
```

Niveaux de visibilité : public et private

Une classe possède des membres de niveaux de visibilité variables.

Il y a en particulier
- **des membres de visibilité publics**
- **des membres de visibilité privés.**

Les membres privés d'une classe ne sont ni connus ni accessibles à l'extérieur de la classe.

Encapsulation des données

Excepté les constantes,
les attributs ne sont généralement pas des membres publics.
L'utilisateur d'une classe n'y a donc pas accès

c'est le principe d'encapsulation des données.

exemple : nous n'avons pas accès aux attributs de la classe Terminal

L'encapsulation des données est une concept clef permettant de s'assurer d'une bonne utilisation de la classe et donc d'éviter de nombreuses erreurs de programmation.
L'accès aux attributs ne peut alors se faire que via l'appel à des méthodes d'instances.

La classe Dimension, qui possèdent deux attributs d'instance publics, est une des très rares exceptions à cette règle

Encapsulation des données : conséquence

**En tant qu'utilisateur de classes déjà définies,
nous ne voyons pas les attributs d'instance.**

**Tant que nous ne définissons pas nos propres classes,
nous n'utilisons donc pas de tels attributs.**

Les membres des classes

Une classe est composée d'éléments appelées les membres de la classe.

Il existe deux types de membres : les fonctions et les attributs.

Les fonctions sont des constructeurs ou des méthodes. Les constructeurs permettent de créer de nouvelles instances.

Les attributs (appelées aussi données membres, variables d'instances) permettent de mémoriser des informations.

Les méthodes et les attributs peuvent être soit d'instance soit de classe (static).

Yannick.Parchemal@parisdescartes.fr

Membres de classe (membres static)

En dehors des constructeurs, il existe des :

Membres d'instance

ils sont toujours appliqués à une instance

- **attributs d'instance : exemple width de la classe Dimension**
- **méthodes d'instance : exemple getNom( ) de la classe Individu**

Membres de classe (static)

ils ne sont pas appliqués à une instance.

- **les attributs de classe : leur valeur est propre à la classe et non à chaque instance particulière .**
  exemple : PI est un attribut static de la classe Math
  ils correspondent à des variables globales de programmation classique
- **les méthodes de classe : elles en sont pas appliquées à une instance**
  exemple : pgcd est une méthode static de la classe Math
  ils correspondent à des fonctions ou procédures de programmation classique

Les méthodes de classe

Les méthodes de classe sont des méthodes non applicables à une instance (on dit aussi méthodes statiques)

Syntaxe : <classe>.<methode>(<paramètres>)

```
// appelle la méthode getDefault de la classe Terminal
// cette méthode rend le premier terminal créé.
Terminal term = Terminal.getDefault( ) ;

// appelle la méthode pgcd de la classe MathUtil
long nb = MathUtil.pgcd(524,118) ;
```

ne pas confondre membres d'instance et membres de classe

```
/** la couleur de fond de la zone de texte de ce terminal devient la couleur 'c'*/
public void setTextAreaColor(Color c){...}
```

term.setTextAreaColor(new Color(200,100,100));

Les membres d'instance sont *toujours* appliqués à une instance

```
/** la couleur de fond de la zone de texte par défaut devient la couleur 'c'
    Au moment de la création d'un terminal, c'est cette couleur qui est choisie */
public static void setDefaultTextAreaColor(Color c){...}
```

Terminal.setDefaultTextAreaColor(new Color(200,100,100));

Les membres de classes sont repérées
dans la documentation par le mot-clé static
Ils ne sont pas appliquées à une instance

Membres d'une classe: les attributs de classe

Les attributs de classe (on dit aussi attributs statiques) sont des variables dont la valeur est la même pour toutes les instances de la classe

Syntaxe :

<classe>.<attribut>

System.out : out est un attribut de classe constant de la classe System dont la valeur est un flux de sortie

Math.PI : PI est un attribut de classe constant de classe de la classe Math

Color.GREEN : GREEN est un attribut de classe constant de la classe Color

Documentation en ligne et utilisation de classes

Pour pouvoir utiliser une classe,
il faut et il suffit de connaître son interface
(la déclaration commentée de tous les membres publics)
consultable grâce à la documentation en ligne

```
// extrait de la documentation en ligne de Terminal
/** crée un Terminal de titre 'titre' et de taille en pixels 'w' * 'h'  */
public Terminal(String titre,int w,int h){...}

/** Affiche un 'message' et rend l'entier lu à partir de ce terminal*/
public int readInt(String message)

/** affiche l'entier 'i' dans ce terminal */
 public void println(int i){...}
```

```
Terminal term = new Terminal("Tri fusion",300,300);
int taille = term.readInt("Donner un entier");
term.println(taille*taille);
```

La documentation des programmes

Pour pouvoir utiliser une classe,
il faut et il suffit de connaître son interface
(la déclaration commentée de tous les membres publics)
consultable grâce à la documentation en ligne

La documentation en ligne peut être générée à partir des fichiers sources par l'utilitaire javadoc.

Cette documentation est organisée et générée de la même manière pour toutes les classes que ce soit les classes de l'API standard ou les classes que nous définissons nous même

Yannick.Parchemal@parisdescartes.fr

A propos des commentaires

Commenter *toujours* les entêtes de fonctions

un bon commentaire permet de pouvoir utiliser la fonction sans consulter le code.

- il indique à l'aide d'une phrase le rôle de la fonction en faisant intervenir **le nom de tous les paramètres**
- il précise **le rôle de chaque paramètre**
- il indique **la signification du résultat retourné**
- il indique **les restrictions sur la valeur des paramètres**

Commenter *si nécessaire* des fragments de codes difficiles (un bon programme en contient généralement peu)

Eviter les commentaires inutiles
a=5; /* a prend la valeur 5 */

les commentaires JAVADOC

Les commentaires javadoc / … */**
sont des commentaires spéciaux
permettant la production automatique de documentation au format html.
Ils sont placés juste avant ce qu'ils commentent.

balises de commentaires JAVADOC de classe

@see <une autre classe>
@author <nom de l'auteur>
@version <n° de version>

balises de commentaires JAVADOC de fonction

@param <nom paramètre> <description>
@return <description>
@exception <nom exception> <description>
@since <n° de version>
@deprecated

Remarque : des balises Html peuvent être incluses dans le commentaire

Contenu d'une documentation javadoc

Description générale de la classe

Description des attributs (FIELD)

Description des constructeurs (CONSTRUCTOR)

Description des méthodes (METHOD)

La description des attributs, des constructeurs et des méthodes publics est donnée
- brièvement en début de document
- en détail dans la suite du document.

Exemple de commentaire javadoc pour une classe

```
/** 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>Savoir si le compte a un solde suffisant</LI>
 *    <LI>Consulter le solde et l'historique des opérations</LI>
 *    <LI>Pouvoir modifier le découvert autorisé</LI>
 *    <LI>Pouvoir sauvegarder sur disque</LI>
 * @author Yannick Parchemal
*/
```

La commande javadoc

GENERATION DE LA DOCUMENTATION

javadoc -d <répertoire où doivent être mis les fichiers générés>
-sourcepath <répertoire(s) de base des fichiers sources>
<nom du paquetage>*

- sourcepath **le(s) répertoire(s) de base des sources**
(s'il y en a plusieurs, séparer par des ; (windows) ou : (linux))

La documentation est ensuite consultable à partir du fichier index.html du répertoire mentionné avec l'option -d

La commande javadoc : exemples d'utilisation

POUR générer la documentation des classes du paquetage essai

```
javadoc -d D:\Dupond\javadev\doc
-sourcepath D:\Dupond\javadev\src
essai
```

par défaut, l'auteur et le n° de version n'apparaissent pas.
Pour voir ces informations dans la documentation,
utiliser les options author et version

```
javadoc -d ..\doc -version -author essai up5.mi.pary.term
```

Les autres options ...

Il existe en fait beaucoup d'options pour javadoc (comme d'ailleurs pour java et javac). Pour en obtenir la liste, tapez simplement le nom de la commande sans paramètre.

Exemple (extraits) :

```
C:\WINDOWS>javadoc
javadoc: No packages or classes specified.
usage: javadoc [options] [packagenames] [sourcefiles] [classnames] [@files]
-overview <file>        Read overview documentation from HTML file
-public                 Show only public classes and members
-protected          Show protected/public classes and members (default)
-package            Show package/protected/public classes and members
-private            Show all classes and members
-help               Display command line options
-doclet <class>         Generate output via alternate doclet
-docletpath <path>      Specify where to find doclet class files
-sourcepath <pathlist>      Specify where to find source files
-classpath <pathlist>  Specify where to find user class files
```

Utilisation de la classe Compte

La classe Compte reprend les fonctionnalités de la classe CompteS et en ajoute de nouvelles qui vont nécessiter l'utilisation de membres de classes.

Elle permet, en plus de la classe Compte :

- **de préciser le moyen de paiement pour effectuer les opérations**
- **de pouvoir gérer un découvert autorisé par défaut attribué lors de la création d'un compte si aucun montant de découvert autorisé n'est spécifié**
- **de pouvoir sauvegarder sur disque**

Yannick.Parchemal@parisdescartes.fr

Méthodes d'instance : enregistrer des opérations

```
/** enregistre une opération de 'montant' Euros sur ce compte
   *  avec le 'moyenPaiement' indiqué
   * @param montant le montant de l'opération
   * @moyenPaiement le moyen de paiement utilisé pour créditer ce compte
*/
public void addOperation(double montant,int moyenPaiement){...}
```

```
Compte c = new Compte("Dupond",600);

c.addOperation(300,Compte. CB);
```

Compte.CB est un attribut de la classe Compte

Les attributs "public" de la classe Compte

4 attributs correspondant à chacun des 4 moyens de paiement.

```
/** la constante pour les paiements par carte bancaire */
  public static final int CB;
/** la constante pour les paiements par virement bancaire */
  public static final int VIREMENT;
/** la constante pour les paiements par cheque */
  public static final int CHEQUE;
/** la constante pour les paiements par cheque */
  public static final int LIQUIDE;
```

final — **Ces quatre attributs sont des contantes (modifieur "final")**

final static — **Ce sont des constantes de classe (modifieur "static") : cela signifie que c'est la même constante pour tous les comptes.**

Deux méthodes de classes : pour obtenir et modifier les découverts autorisés par défaut attribués lors de la création des comptes

```
/** @return le decouvert autorise par défaut attribué lors de la création des comptes */
public static double getDecouvertAutoriseParDefaut( ) {...}

/** le decouvert autorise  par défaut lors de la création des comptes
   * devient égal à 'decouvertAutorise'*/
public static void setDecouvertAutoriseParDefaut(double decouvertAutoriseParDefaut) {...}
```

Ces deux méthodes ne s'appliquent pas à un compte : elle modifie une caractéristique propre à la classe : ce sont des méthodes de classes (ou méthodes "static")

```
Compte.setDecouvertAutoriseParDefaut(2000);
Compte c = new Compte("Dupond",600);
System.out.println(c.getDecouvertAutorise( ));/*2000*/
```

Membres de classes et membres d'instances de la classe Compte

Méthodes d'instances

c.setDecouvertAutorise(2000);
setDecouvertAutorise est une méthode d'instance:
elle s'applique à une instance

Méthodes de classes

Compte.setDecouvertAutoriseParDefaut(2000);
setDecouvertAutoriseParDefaut est une méthode de classe
elle ne s'applique pas à une instance particulière de la classe

Attributs de classes

CB est un attribut de classe (constant) de la classe Compte
System.out.println(Compte. CB);

Méthodes d'instances et de classes: Sauvegarde et restauration des comptes sur disque

```
/** enregistre ce compte dans un fichier du répertoire courant
  * @return true si le compte a été sauvegardé, faux sinon */
public boolean sauvegarder( ){}

/** @return le compte de 'titulaire', crée un nouveau compte s'il n'est pas trouvé sur disque */
public static Compte charger(String titulaire){}
}
```

```
Compte c = Compte.charger("Dupond");
…
c.sauvegarder( );
```

sauvegarder est une méthode d'instance : elle est appliquée à une instance
charger n'est pas appliquée à une instance : c'est une méthode de classe.

Extrait de la documentation de la classe Compte

```
/** @return le decouvert autorise  pour ce compte */
public double getDecouvertAutorise( ) {...}
/** le decouvert autorise  pour ce compte devient égal à 'decouvertAutorise'*/
public void setDecouvertAutorise(double decouvertAutorise) {...}

/** @return le decouvert autorise par défaut attribué lors de la création des comptes */
public static double getDecouvertAutoriseParDefaut() {...}
/** le decouvert autorise  par défaut lors de la création des futurs comptes
   * devient égal à 'decouvertAutorise'*/
public static void setDecouvertAutoriseParDefaut(double decouvertAutoriseParDefaut) {...}


/** enregistre ce compte dans un fichier du répertoire courant */
public boolean sauvegarder( ){...}
/** @return le compte de 'titulaire', crée un nouveau compte s'il n'est pas trouvé sur disque */
public static Compte charger(String titulaire){...}
}
```

Affectations et passages de paramètres

Les affectations et les passages de paramètres se passent différemment pour les types simples d'une part et les objets d'autre part.

Pour les types simples, l'affectation est avec recopie et le passage de paramètres par valeur.

Pour les objets, l'affectation se fait sans recopie et le passage de paramètres par référence.

Il n'y a recopie d'objets que sur demande explicite du programmeur

Yannick.Parchemal@parisdescartes.fr

Affectations à des variables de types simples

L'affectation à des variables de types simples se fait avec recopie de la valeur

C'est donc un affectation "classique"
(comme en C, C++, C#, Pascal, Fortran, Php etc …)

```
int i = 7;
```
i 7

```
int j = i;
/* copie
de i dans j*/
```
i 7
j 7

j a maintenant comme valeur une copie de i

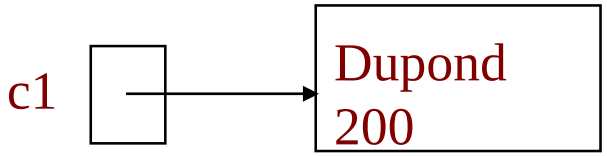
```
i=i+1;
```
i 8

j 7

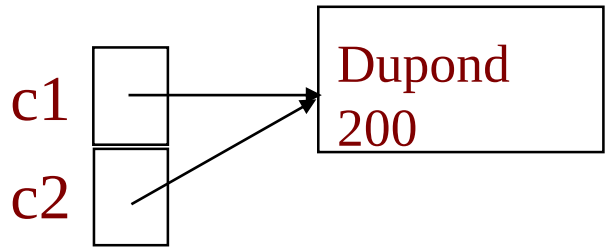
mais les deux variables ne sont pas liées

Affectation sans recopie pour les objets

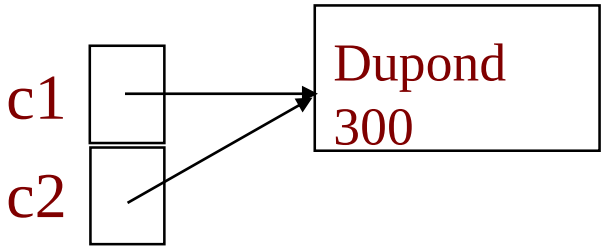
```
Compte c1 = new Compte("Dupond",200);
```




```
Compte c2 = c1;
/*c1 et c2 référencent
   le même objet */
```



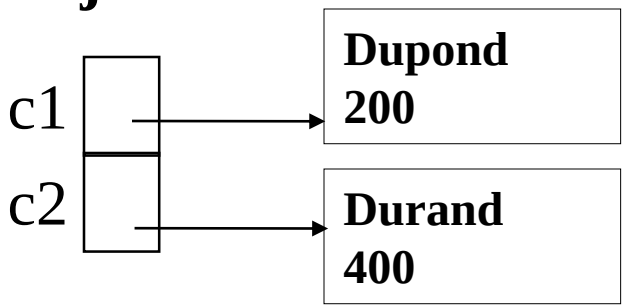

```
c1.addOperation(100);
/* on aurait pu écrire
c2.addOperation(100);
*/
```



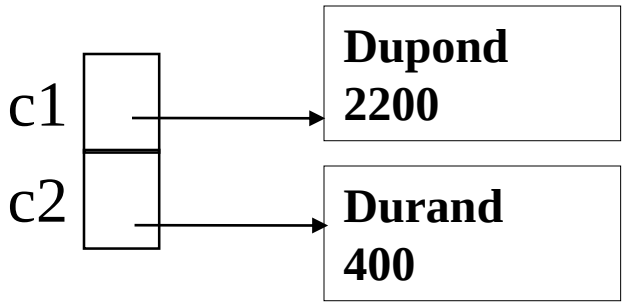
Objets et référence sur des objets

Avec Java, on manipule en fait des références sur les objets et jamais directement les objets eux-mêmes

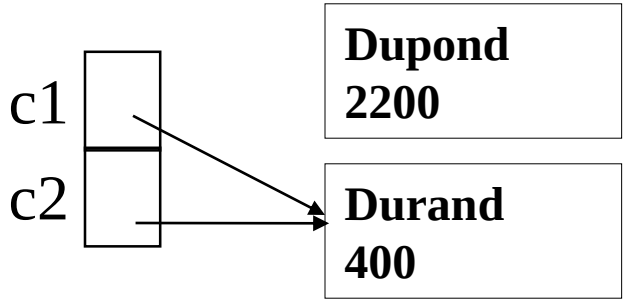
```
Compte c1,c2;
c1 = new Compte("Dupond",200);
c2 = new Compte("Durand",400);
```




```
c1.addOperation(2000);
```




```
c1=c2;
```

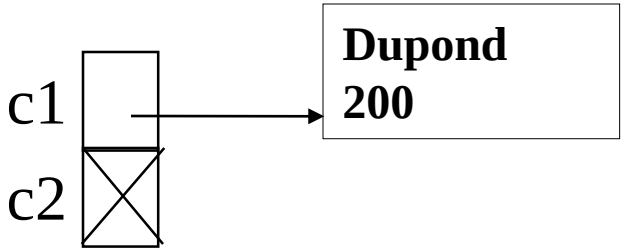


null signifie "absence de référence"

null signifie "absence de référence"

```
Compte c1,c2;
c1 = new Compte("Dupond",200);
c2=null;
```

null




```
System.out.println(c1.getSolde());
```

200

```
System.out.println(c2.getSolde());
```

java.lang.NullPointerException

Appliquer une méthode à une référence null entraîne une erreur à l'exécution.

null est la valeur par défaut des variables de type classe.

Passage par valeurs pour les types simples

Passage par valeurs pour les types simples

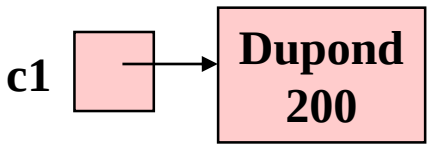
```
public class TestPassage {
private static int f(int j){
      j=j+1;
     return(j);
}
public static void main(String[] tArg) {
      int i = 9;
       System.out.println("i="+i);
      int r = f(i);
      System.out.println("i="+i+" r="+r);
}}
```

i=9
i=9 r=10

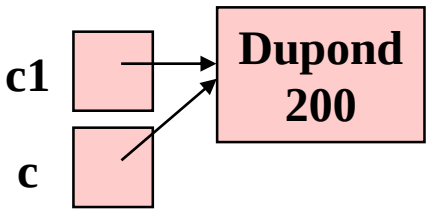
Passage par référence pour les objets

```
public class TestPassage {

private static void f(Compte c){
   c.addOperation(100);
}

public static void main(String[] tArg) {
      Compte c1 = new Compte("Dupond",200);
      System.out.println(" c1:"+c1.getSolde());
      f(c1);
      System.out. println(" c1:"+c1.getSolde());
}}
```

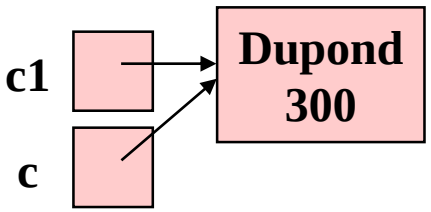
c1:200
c1:300



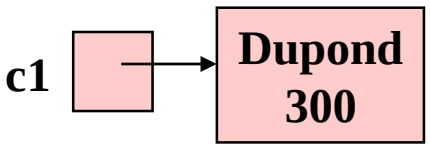
Initialisation de c1



Appel de f



Instruction de f

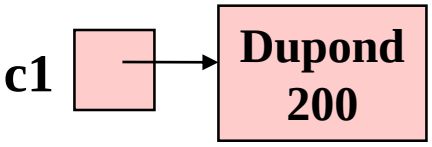


Après le retour de f

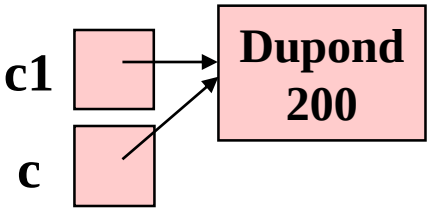
Passage par référence pour les objets

```
public class TestPassage {
private static void  g(Compte c){
   c=new Compte("Durand",400);
}
public static void main(String[] args) {
      Compte c1 = new Compte("Dupond",200);
      System.out.println(" c1:"+c1.getSolde( ));
      g(c1);
      System.out. println(" c1:"+c1.getSolde( ));
}}
```

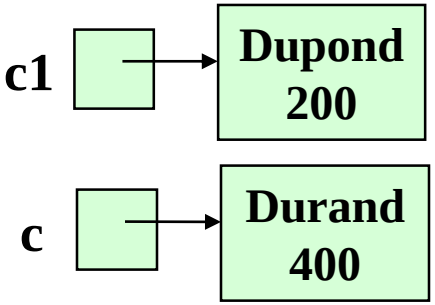
c1:200
c1:200



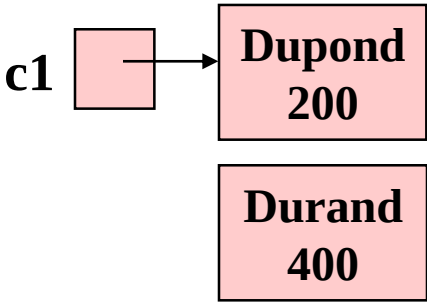
Initialisation de c1



Appel de g



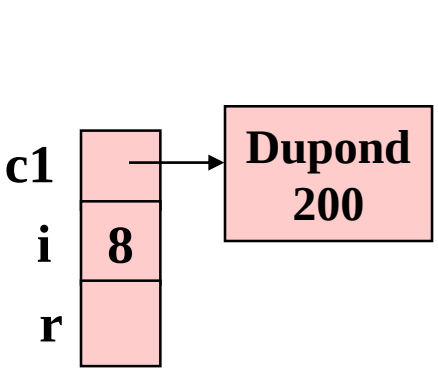
Instruction de g



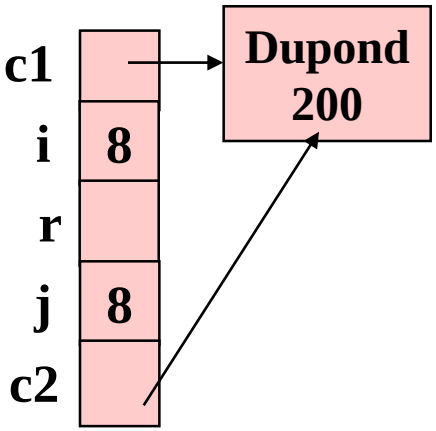
Après le retour de g

Passages par valeur et référence : un autre exemple

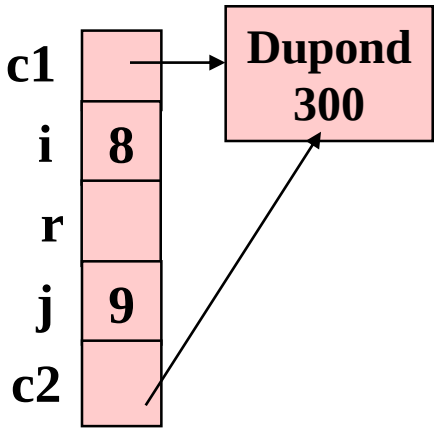
```
public class TestPassage {
public static int h(int j,Compte c2){
        j=j+1;
        c2.addOperation(100);
        return(j);
}
public static void main(String[] args) {
        Compte c1 = new Compte("Dupond",200);
        int i = 8;
        int r = h(i,c1);
        }}
```



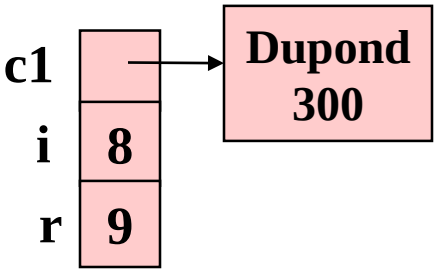
Initialisations



Appel de h



Instructions de h



Après le retour de h

JAVA et les pointeurs

PAS DE MANIPULATION EXPLICITE DE POINTEURS

MANIPULATION IMPLICITE PERMANENTE :
les valeurs objets sont en fait des références sur les objets

Résumé:
Les affectations avec les variables de type "classe"
sont des affectations sans recopie de l'objet affecté
Les passages de paramètres pour les variables de type classe
sont des passages par référence

Les tableaux étant des objets, les mêmes règles s'appliquent pour eux

Le ramasse-miettes

JAVA gère automatiquement la mémoire:
un "garbage collector" (ramasse miettes)détecte les objets inutilisables et les supprime.

Le programmeur n'a pas à gérer la mémoire
en rendant explicitement les objets devenus inutiles

Le garbage collector fonctionne en tâche de fond :
un objet peut être effectivement supprimé longtemps après qu'il ne soit devenu inutilisable.

le ramasse miettes peut être appelé explicitement par System.gc().

Yannick.Parchemal@parisdescartes.fr

Objets inutilisables

un objet est inutilisable s'il n'est plus référencé

exemple

On considère l'objet créé par l'instruction suivante
et on étudie le nombre de références sur cet objet

```
Date d = new Date( ); /*   1 (d)  */
Date d2 = d; /* 2 (d d2) */
Date d3 = new Date( ); /* 2 (d d2) */
d3 = d2; /*3 (d d2 d3) */
d2 = new Date(); /* 2 (d d3) */
d3 = null; /* 1 (d) */
d = d2;  /* 0 ( ) */   /* l'objet est maintenant inutilisable */
```

Classes et tableaux

Un tableau est un objet instance d'une classe
qui est construite automatiquement
(une classe pour chaque type d'éléments de tableau)

```
Date [ ] tDate = new  Date[10];
tDate[5]=new Date();
System.out.println(tDate[5].getClass());
System.out.println(tDate.getClass());
```

class java.util.Date
class [Ljava.util.Date;

Les tableaux
nombre d'éléments: l'attribut length

On peut accéder au nombre d'éléments d'un tableau grâce à l'attribut "length"

```
String [] args= new String[12];
System.out.println(args.length);
// args.length désigne la longueur
// du tableau args
```

12

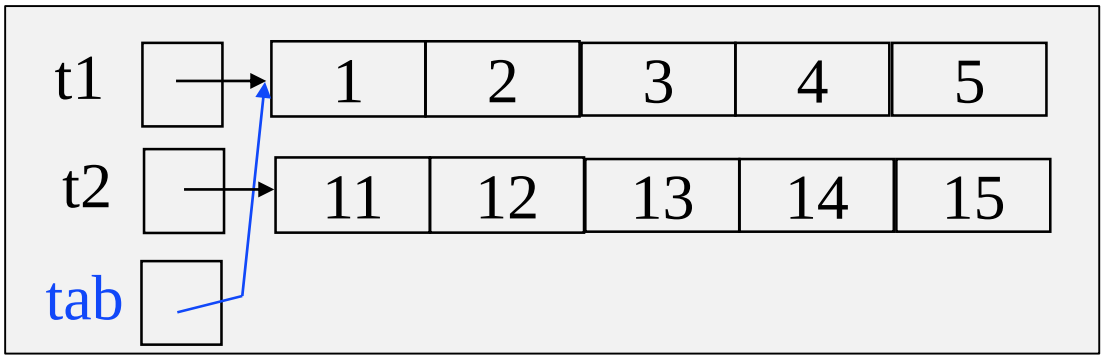
On ne peut pas modifier la valeur de "length" : c'est un attribut constant

~~args.length = 4;~~

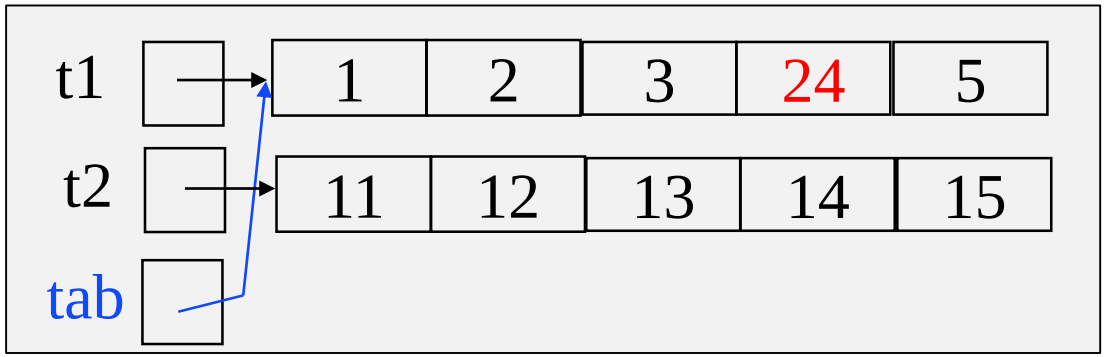
TABLEAUX : affectation

Les tableaux sont des objets :
l'affectation d'un tableau à une variable tableau
n'entraîne donc **pas de recopie**!

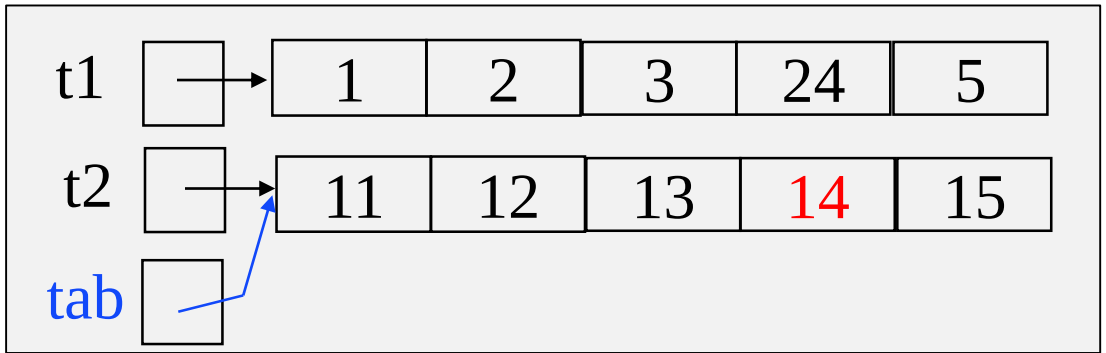
```
int [] t1 = {1,2,3,4,5};
int [] t2 = {11,12,13,14,15};
int [] tab = t1;
```




```
t1[3]=24;
System.out.println(tab[3]);
```




```
tab = t2;
System.out.println(tab[3]);
```



Valeur par défaut des éléments d'un tableau

La valeur par défaut d'un élément du tableau est la valeur par défaut du type des éléments de ce tableau.

```
int [] t1 = new int[10];

System.out.println(t1[4]);
```

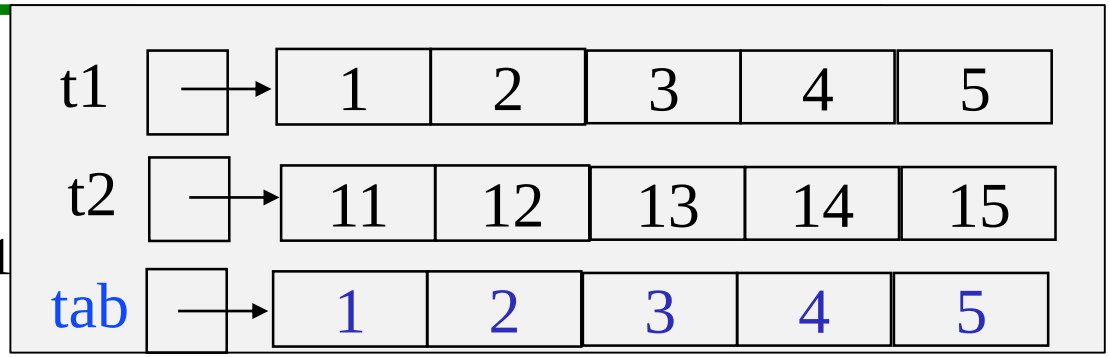
0

```
Date [] t2 = new Date[10];

System.out.println(t2[4]);
```

null

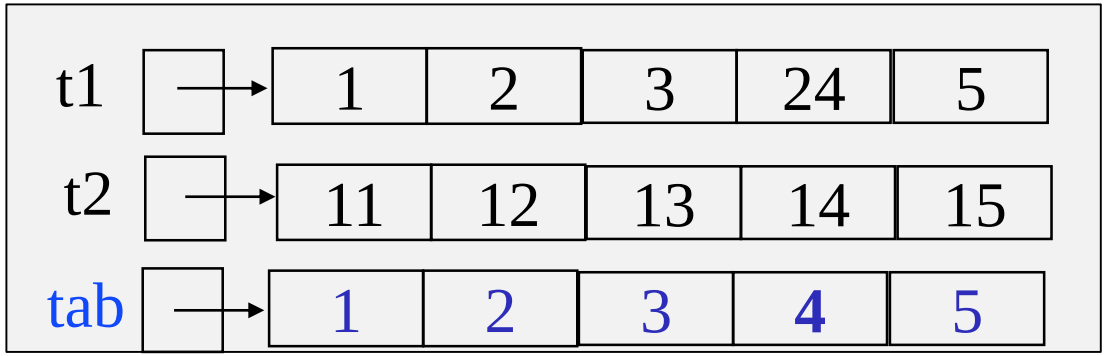
TABLEAUX : copie

```
int [ ] t1 = {1,2,3,4,5};
int [ ] t2 = {11,12,13,14,15};
// on veut que tab soit une copie de t1
int [] tab = new int [t1.length];
for (int i=0;i<t1.length;i++)
      tab[i]=t1[i];
```



// la même chose en plus rapide : **int[]tab = (int [ ]) t1.clone()**

```
t1[3]=24;
System.out.println(tab[3]);
// affichage de 4 : t1 et tab sont bien 2 tableaux différents
```



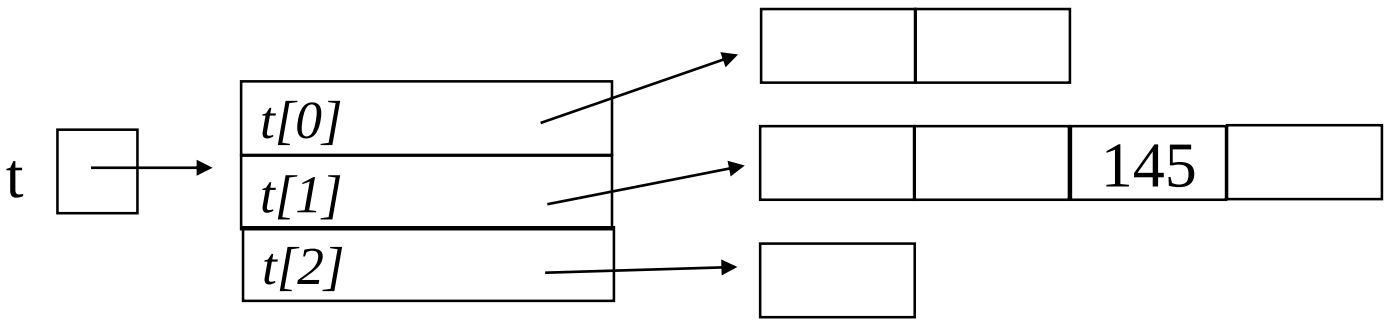
Tableaux de tableaux

Les éléments d'un tableau peuvent être des tableaux

int [] [] t; /* t est un tableau de tableaux d'entiers */

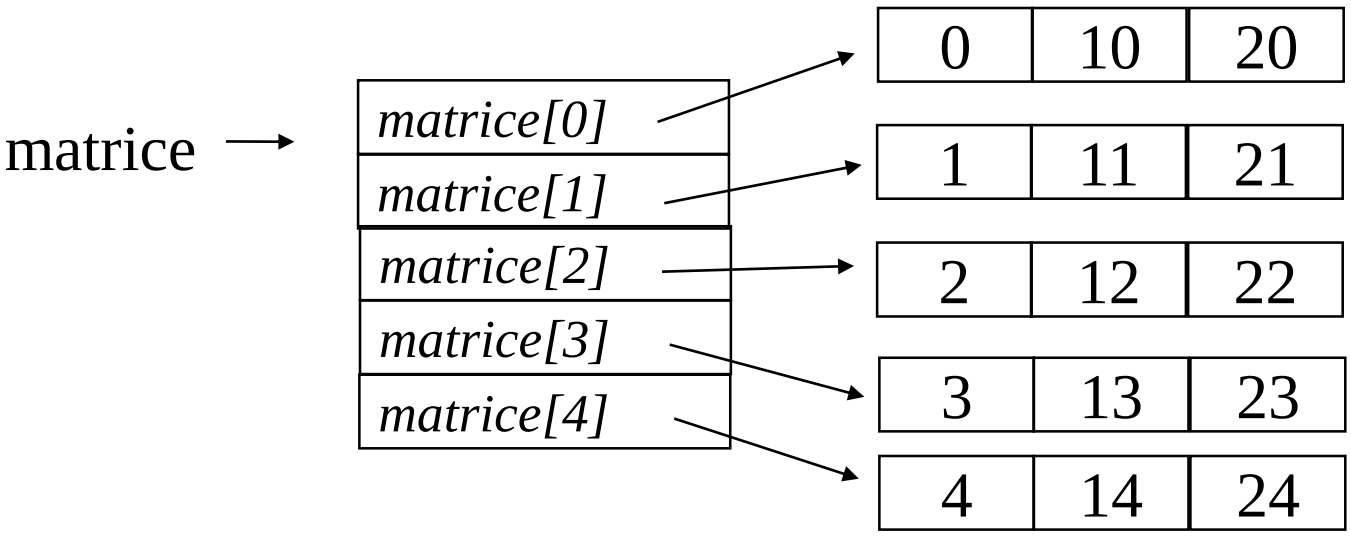
t= new int [3] []; /* t est un tableau de 3 éléments (qui sont des tableaux d'entiers)*/
t[1] = new int [4];
t[0] = new int [2];
t[2] = new int[1];

t[1][2]=145;



Tableaux de tableaux: les matrices

```
int [][] matrice;  // matrice est un tableau de tableau d'entiers
matrice = new int[5][3]; // matrice 5x3
for (int i=0; i< matrice.length;i++)  /* matrice.length == 5*/
    for (int j=0;j<matrice[i].length;j++)   /* matrice[i].length == 3*/
matrice[i][j]=i+10*j;
```



Tableaux de tableaux: initialisation

```
int [] [] tab = {{7,3,8},{9}};
```

équivalent à:

```
int [] [] tab = new int[2][];
tab[0] = new int [3];
tab[1] = new int [1];
tab[0][0]=7;tab[0][1]=3;tab[0][2]=8;
tab[1][0]=9;
```

tableaux de tableaux

Les tableaux de tableaux ne sont que rarement utilisés sauf pour représenter des matrices de nombres pour des algorithmes numériques.

Exemple des groupes d'élèves:

A des tableaux de tableaux de notes, on préfère des tableaux d'élèves, à chaque élève étant associé un tableau de notes.

Utilisation de l'API

la classe java.util.GregorianCalendar

la classe java.util.TimeZone

Yannick.Parchemal@parisdescartes.fr

La classe GregorianCalendar

Nombre de millisecondes depuis le 1er janvier 1970 OH GMT

universels mais pas pratique

calendrier

il en existe beaucoup

calendrier grégorien

c'est le nôtre

nombre de millisecondes depuis le 1/1/1970 GMT+00 ⟷ 23 Aout 2013 13H10 GMT+2 à Paris
30 Sravana 1935 16H40 GMT +5H30 à Bombay

Voir par exemple http://www.calendarhome.com/converter/

Le calendrier grégorien (depuis 1582)

année de 365 ou 366 jours (bissextile)

12 mois	janvier :	31 jours
	février :	28 jours ou 29 (années bissextile)
	mars :	31 jours
	avril :	30 jours
	mai :	31 jours
	juin :	30 jours
	juillet :	31 jours
	août :	31 jours
	septembre :	30 jours
	octobre :	31 jours
	novembre :	30 jours
	décembre :	31 jours

les années bissextiles sont les années multiples de 4
à l'exception des années multiples de 100 non multiples de 400

1 jour = 24H 1H = 60 mn 1 mn = 60 sec. 1 sec = 1000 millisec

GregorianCalendar

JANVIER 2021

L	M	Me	J	V	S	D
			1	2	3	
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

FEVRIER 2021

L	M	Me	J	V	S	D
1	2	3	4	5	6	7
8	**9**	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28

MARS 2021

L	M	Me	J	V	S	D
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Permet

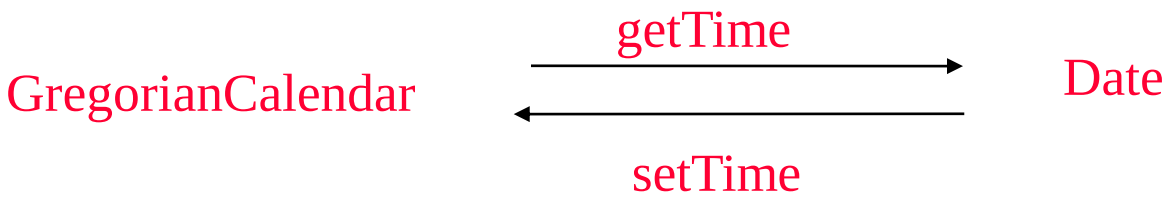
- de définir et de manipuler des dates dans le cadre du calendrier grégorien
- d'obtenir la **Date** équivalente

La classe GregorianCalendar

extraits de la documentation en ligne

```
package java.util;
public class GregorianCalendar extends Calendar {

/** Constructs a default GregorianCalendar using the current time in the default time zone with
the default locale. */
public GregorianCalendar( ) {…}

/** Constructs a GregorianCalendar with the given date set in the default time zone with the
default locale.
Parameters:
   year - the value used to set the YEAR time field in the calendar.
   month - the value used to set the MONTH time field in the calendar. Month
                value is 0-based.   e.g., 0 for January.
   date - the value used to set the DATE time field in the calendar. */
public GregorianCalendar(int year, int month, int date) {…}

/** Constructs a GregorianCalendar with the given date and time set for the default time zone
with the default locale. */
public GregorianCalendar(int year, int month, int date, int hour, int minute,int second) {…}
```

GregorianCalendar et Date




```
/** rend la Date correspondant à ce gc */
public Date getTime( ) {…}
/** la date courante de ce gc devient la date correspondante à d*/
public void setTime(Date d){…}
```

```
GregorianCalendar gc = new GregorianCalendar(2013,Calendar.AUGUST,23,11,0,0);
Date d = gc.getTime();
System.out.println(d);
gc.setTime(new Date(1000000000000L));
System.out.println(gc.getTime());
```

constante correspondant au mois d'aout

```
Fri Aug 23 11:00:00 CEST 2013
Sun Sep 09 03:46:40 CEST 2001
```

Champs définissant une date dans le calendrier

Calendar.YEAR
Calendar.MONTH
Calendar.DATE
Calendar.HOUR_OF_DAY
Calendar.MINUTE
Calendar.SECOND
Calendar.MILLISECOND

```
/**Gets the value for a given time field of this calendar
@ param   field    the given time field.
@ return   the value for the given time field.*/
public final int get(int field){…}
```

```
GregorianCalendar cal = new GregorianCalendar(2013,Calendar.AUGUST,23,11,15,0);
System.out.println(" YEAR   = "+ cal.get(Calendar. YEAR ));
System.out.println(" MONTH = "+ cal.get(Calendar. MONTH ));
System.out.println(" HOUR_OF_DAY = "+cal.get(Calendar. HOUR_OF_DAY ));
System.out.println(" MINUTE = "+cal.get(Calendar. MINUTE ));
System.out.println(" SECOND =  "+cal.get(Calendar. SECOND));
System.out.println(" MILLISECOND =  "+cal.get(Calendar. MILLISECOND ));
```

```
YEAR   = 2013
MONTH = 7
HOUR_OF_DAY = 11
MINUTE = 15
SECOND =  0
MILLISECOND =  0
```

La classe GregorianCalendar

add : pour modifier le champ d'une date d'un calendrier d'un nombre donné

```
GregorianCalendar gc = new GregorianCalendar(2014,Calendar.OCTOBER,9,11,0,0);
gc.add(Calendar.DATE,1);
System.out.println(gc.getTime());

gc.add(Calendar. HOUR_OF_DAY,1);
System.out.println(gc.getTime());

gc.add(Calendar.DATE,+30);
System.out.println(gc.getTime());
```

```
Fri Oct 10 11:00:00 CEST 2014
Fri Oct 10 12:00:00 CEST 2014
Sun Nov 09 12:00:00 CET 2014
```

La méthode add de Gregoriancalendar

extraits de la documentation en ligne

```
/**Adds the specified (signed) amount of time to the given time field,
*      based on the calendar's rules.
* @ param   field         the time field.
* @ param   amount     the amount of date or time to be added to the field.
*/
public void add(int field, int amount){…}
```

```
public boolean after(Object g){ … }
public boolean before(Object g) { … }
public boolean equals(Object g) { … }
```

Zone horaire

11/08/1999 11H : HEURE FRANCAISE

HEURE FRANCAISE (Central European Time):
Heure d'hiver (CET) : GMT +1
Heure d'été (CEST): GMT +2 du dernier dimanche de mars 1H GMT
au dernier dimanche d'octobre 1H GMT

Si la machine Java est bien configurée,
la zone horaire par défaut est la bonne !

La classe TimeZone contient les définitions des zone horaires
et permet de modifier la zone horaire par défaut.

Chaque zone horaire a un ID
En France, c'est "Europe/Paris"
A Dakar, c'est "Africa/Dakar"
voir http://www.timeanddate.com/time/abbreviations.html

TimeZone

Pour connaître la zone horaire par défaut:

```
TimeZone tGMT = TimeZone.getDefault();
System.out.println("Zone horaire par defaut :"+tGMT.getID());
```

```
Zone horaire par defaut :Europe/Paris
```

Pour obtenir une zone horaire:

```
TimeZone maZoneHoraire = TimeZone.getTimeZone("Africa/Dakar");
```

Pour modifier la zone horaire par défaut

```
TimeZone.setDefault(maZoneHoraire);
```

La classe java.util.TimeZone

```
Date date = new GregorianCalendar(2013,Calendar.AUGUST,26,11,15,0).getTime();
TimeZone tGMT = TimeZone.getDefault();
System.out.print("Zone horaire : "+tGMT.getID());
System.out.println("  d="+date);

TimeZone zoneHoraire = TimeZone.getTimeZone("Africa/Dakar");
TimeZone.setDefault(zoneHoraire);
System.out.print("Zone horaire  : "+TimeZone.getDefault().getID());
System.out.println("  d="+date);

TimeZone.setDefault(TimeZone.getTimeZone("Australia/Sydney"));
System.out.print("Zone horaire  : "+TimeZone.getDefault().getID());
System.out.println("  d="+date);
```

Zone horaire : Europe/Paris d=Mon Aug 26 11:15:00 CEST 2013
Zone horaire : Africa/Dakar d=Mon Aug 26 09:15:00 GMT 2013
Zone horaire : Australia/Sydney d=Mon Aug 26 19:15:00 EST 2013

Héritage et liaison dynamique

Objet

Classe

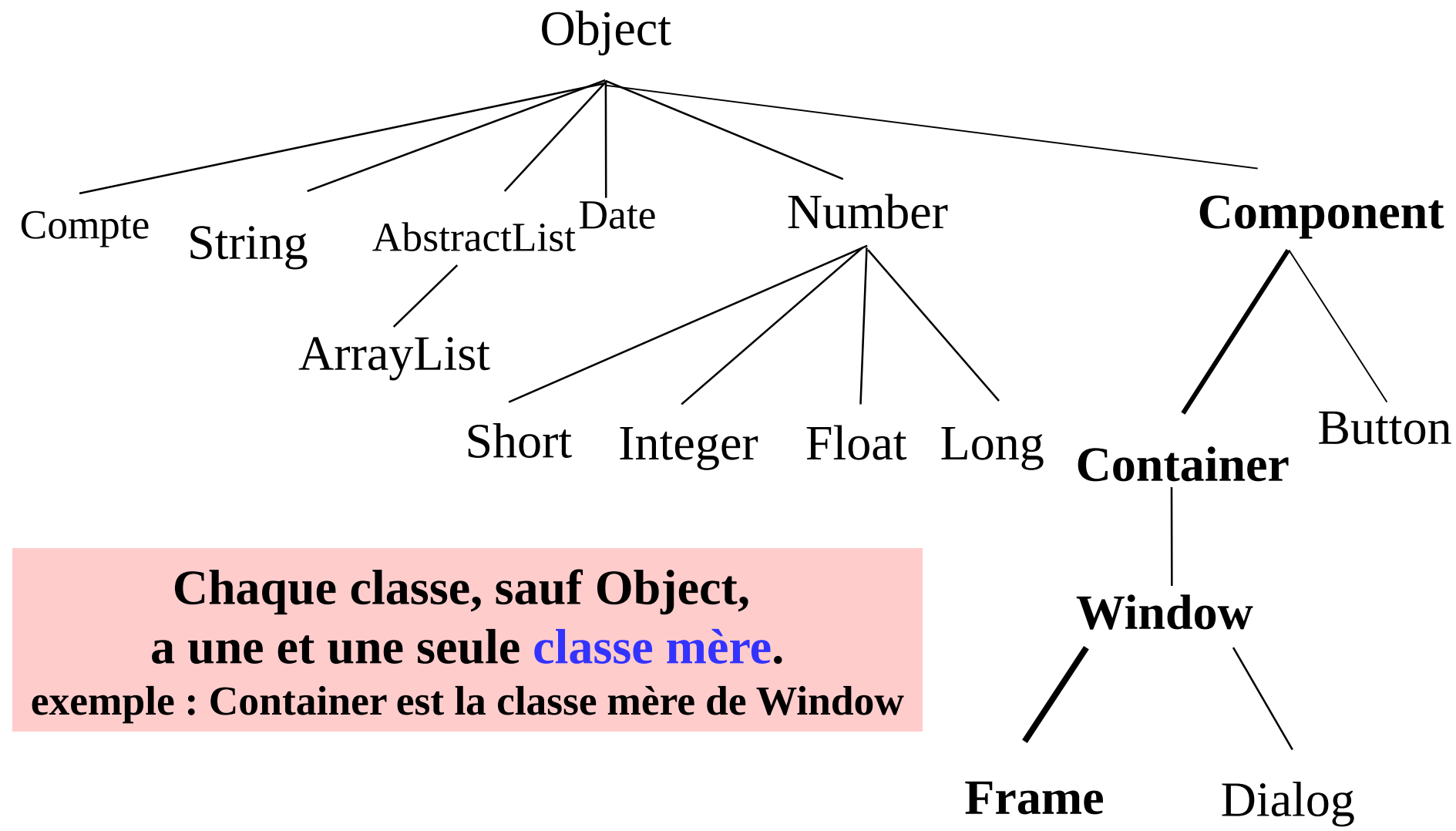
Encapsulation des données

*** Héritage**

Liaison dynamique (Polymorphisme)

Yannick.Parchemal@parisdescartes.fr

L'arborescence de dérivation de classes



Chaque classe, sauf Object,
a une et une seule classe mère.
exemple : Container est la classe mère de Window

HERITAGE

HERITAGE:
Une classe hérite de tous les membres de sa classe mère

Les membres publics hérités sont utilisables comme s'ils étaient définis dans la classe.

exemple :

La classe Component hérite des membres de la classe Object

La classe Container hérite des membres de la classe Component (et donc indirectement de ceux de la classe Object)

la classe Frame hérite des membres de la classe Window

HERITAGE : exemple de la classe Frame

La méthode void setBackground(Color c) est définie dans la classe Component

```
public class Component extends Object {
……
/** la couleur de fond du composant devient égale à 'c' */
public void setBackground(Color c){…}
```

Elle n'apparaît pas dans la documentation de la classe Frame car elle est héritée de la classe Component.
Elle est cependant applicable à toute instance de la classe Frame.

```
Frame frame = new Frame("Une fenetre");
frame.setBackground(java.awt.Color.yellow);
frame.setSize(300,200);
frame.setVisible(true);
```

Rq: il en est par ailleurs de même pour les méthodes setSize et setVisible

Partie héritée, partie spécifique

Exemple (incomplet) de la classe Frame

partie héritée
de la classe Object

```
… …
public Object()
protected Object clone();
public boolean equals(Object  obj);
protected void finalize();
public final Class getClass();
public String toString();
```

partie héritée
de la classe Component

```
public void setBackground(Color c){…}
public void setSize(int w,int h){…}
public void setVisible(boolean visible){…}
```

partie spécifique
de la classe Frame

```
public void setTitle(String titre){…}
```

Classe mère et classe fille

LA CLASSE MERE
ou la super-classe

Component est la classe mère de Container

On dit aussi : Component est **la** super classe de Container

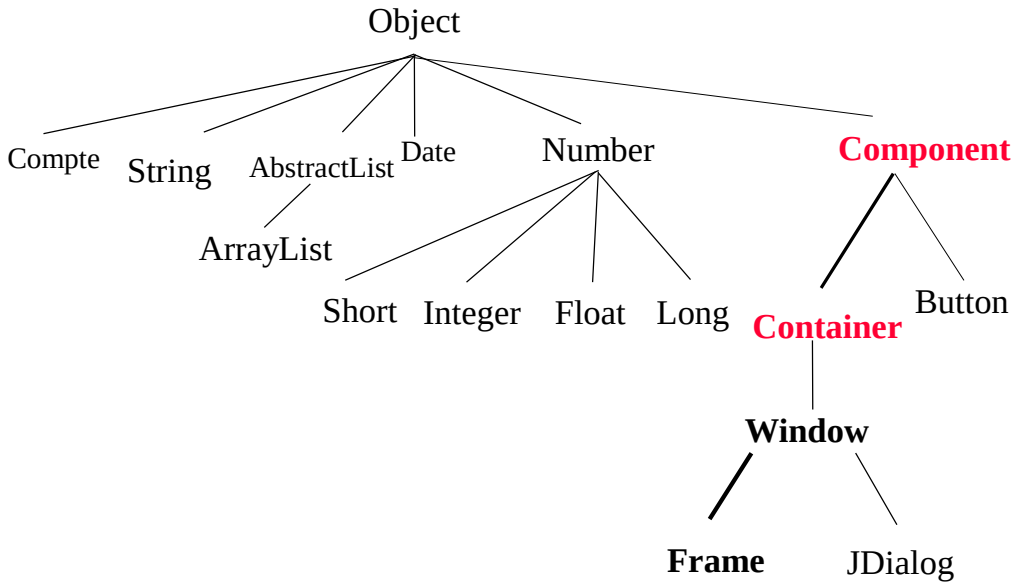
CLASSE FILLE
ou dérive directement de
ou **extends (java)**

Container est une classe fille de Component

On dit aussi :

Container dérive directement de Component

public class Container extends Component ...



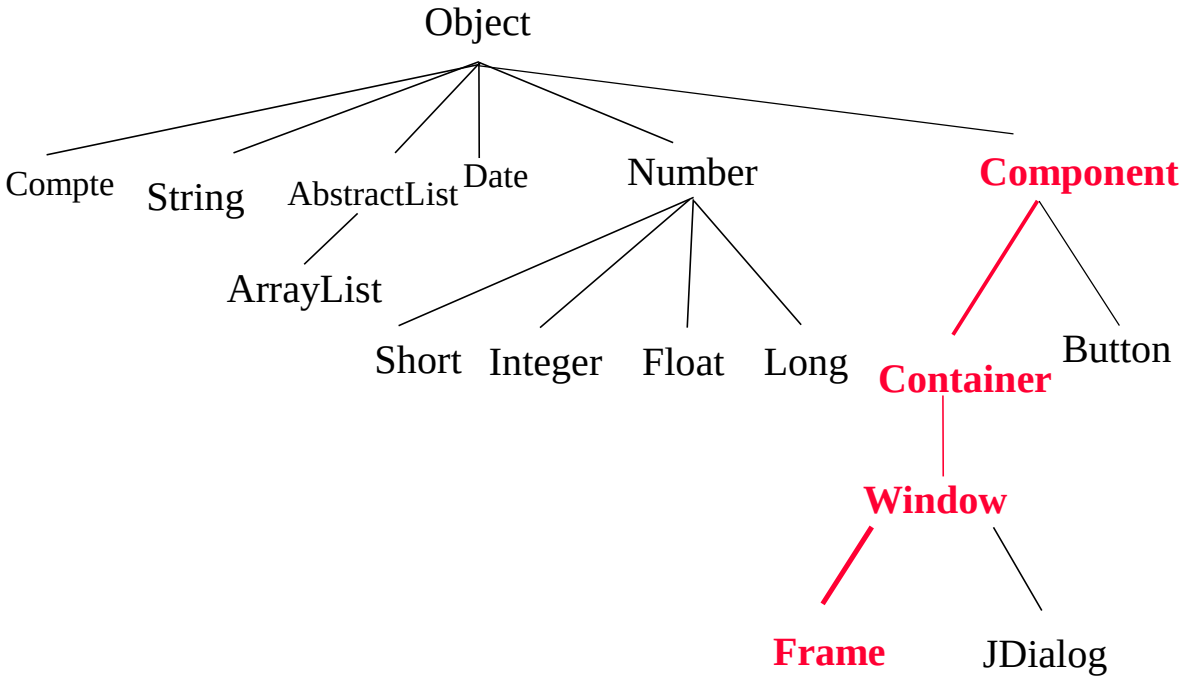
Classe ancêtre et classe dérivée

CLASSE ANCETRE
ou une super-classe

Component est une classe ancêtre de Frame
On dit aussi : Component est **une** super classe de Frame

CLASSE DERIVEE
ou sous-classe

Frame est une classe dérivée de Component
On dit aussi : Frame est une sous-classe de Component



Instance directe et indirecte

Frame f = new Frame("Mon application");

f est une instance directe de Frame
f est une instance indirecte de Window,Component et de Object

Un objet o de classe C est une **instance directe** de C

Si C est une classe dérivée de S, o est une **instance indirecte** de S

Tous les objets sont des instances de la classe Object

Toutes les classes (sauf Object) sont des classes dérivées de la classe Object

L'opérateur instanceof

syntaxe : <instance> instanceof <classe>
teste si <instance> est une instance (directe ou indirecte) de <classe>

```
Object d = new Date( );
System.out.println(d instanceof Date);//true car la classe de l'objet est la classe Date
System.out.println(d instanceof Object);//true car une Date est un Objet
System.out.println(d instanceof String);//false car une date n'est pas une chaine

Object r= new Object( );
System.out.println(r instanceof Object);//true
System.out.println(r instanceof Date);
        //false car une instance directe de Object n'est pas une Date
```

null instanceof C == false
Si x != null alors x instanceof Object est toujours vrai

La classe Object

Les membres de la classe Object sont hérités par toutes les classes

```
public  class  java.lang.Object
{
   // Constructeurs
   public Object( );
   // Quelques méthodes
   protected Object clone( );        // rend une recopie de cet objet
   public boolean equals(Object  obj);// teste si obj est égal à cet objet
   protected void finalize( );       // Appelé avant la destruction de cet objet
   public final Class getClass( );       // rend la classe de cet objet
   public String toString( );        // <nomClasse>@xxxxxx

   public int hashCode( );         // rend un code associé à l'objet
   ….
}
```

La classe Object : la méthode String toString ( )

La méthode String toString( ) est définie dans la classe Object .
Elle retourne une chaîne : <nomDeLaClasse>@code
Toutes les classes héritent de cette méthode(car toutes les classes dérivent de Object)
Cette méthode étant publique, elle est applicable à toute instance.

```
Random gen = new Random( );
String s = gen.toString( );
System.out.println(s);
```

java.util.Random@75f2709a

Héritage et méthodes "redéfinies"

Une méthode héritée par une classe peut être redéfinie dans cette classe.
C'est alors cette nouvelle méthode qui est appliquée aux instances de la classe.

Exemple avec la méthode toString

La méthode String toString( ) est souvent redéfinie dans les classes dérivées fournissant un résultat plus adapté.

```
// extrait de la définition de la classe Date
/** retourne une chaîne correspondant à cette date */
public String toString( ){
        ...
        }
```

```
Date date = new Date(0);
String s = date.toString();
System.out.println(s);
```

```
Thu Jan 01 01:00:00 CET 1970
```

La classe Object : la méthode void finalize( )

Permet d'effectuer certaines tâches
juste avant que le ramasse-miette ne supprime un objet inutile

Telle qu'elle est définie dans la classe Object,
elle ne fait rien mais elle peut-être redéfinie
dans n'importe quelle classe dérivée.

elle est redéfinie pour libérer des ressources "externes":
fichier à fermer, fenêtre à supprimer

Transtypage : exemple

1

Object d = new Date( ); // Pas de problème : une date est un objet

2

Date d = new String( ); // Impossible : une date n'est une chaîne !

3

Date d = new Object( );// Impossible : une instance directe de Object n'est pas une date !

4

```
Object d = new Date( );
…
Date d1 = d; // Refusé par le compilateur car tous les objets ne sont pas des dates

Date d1 = (Date) d; // Accepté : je "promets" au compilateur que d est une Date.
```

Le transtypage permet de préciser au compilateur la classe d'un objet.

Transtypage : fonctionnement

Object d = new String( );

…

Date d1 = d; // Refusé par le compilateur car tous les objets ne sont pas des dates

Date d1 = (Date) d;

Accepté à la compilation :
je "promets" au compilateur que d est une Date

System.out.println(d1.getTime( ));

Vérifié et refusé à l'exécution

```
ERROR: java.lang.ClassCastException: java/lang/String
Press any key to continue...
```

Le transtypage n'est pas une conversion d'un objet d'un type vers un objet d'un autre type : l'objet n'est pas modifié.
Il s'agit plutôt d'une promesse faite au compilateur, vérifiée à l'exécution par l'interpréteur.

Polymorphisme

Objet

Classe

Encapsulation des données

Héritage

*** Polymorphisme**

Yannick.Parchemal@parisdescartes.fr

Polymorphisme (1)

```
int i = term.readInt("Donner une valeur entière");

Object obj = new Date( );

if (i==1) obj = new Compte("Dupond",400,400);

String s = obj.toString( ); // accepté car toString est définie dans la classe Object
                  // la méthode effectivement exécutée sera ici
                                    // celle de la classe Date ( i!=1 )
                                    // ou celle de la classe Compte ( i == 1 )
System.out.println(s);
```

La méthode toString qui est effectivement appelée
est la méthode toString de la classe de obj au moment de l'exécution.
Elle peut prendre, au moment de l'exécution, plusieurs formes,
d'où le nom de polymorphisme.

Polymorphisme (2)

LIAISON DYNAMIQUE

La méthode effectivement appelée est déterminée
au moment de l'exécution
en fonction de la classe effective de l'objet
et non en fonction de la "classe apparente" au moment de la compilation.

```
Object obj = new Date();
...
String s1 = obj.toString();
    /* c'est la méthode toString( ) de la classe Date qui est utilisée*/
```

Polymorphisme : exemple

```
Object [ ] tab  = new Object[3];
tab[0]= new Compte("Dupond",200);
tab[1]= "une chaîne";
tab[2]= new Date(0);

for (int i=0;i<tab.length;i++){
   String s = tab[i].toString();
   System.out.println(s.toUpperCase());
   }
```

La méthode toString qui est effectivement appelée
n'est pas la même selon les éléments du tableau.
C'est la méthode toString de la classe de l'élément qui est appliquée.
(soit la méthode redéfinie, soit la méthode héritée)

Une fonction utilisant le polymorphisme

```
/** affiche tous les éléments du tableau 'tab' */
public static void afficherEnMajuscule(Object [ ] tab){
   for (int i=0;i<tab.length;i++){
      String s = tab[i].toString();
      System.out.println(s.toUpperCase());
     }
}
```

La fonction afficherEnMajuscule permet d'afficher tous les éléments du tableau tab en majuscule par l'utilisation de la méthode toString.

La classe java.util.ArrayList

réalise l'implémentation de tableaux de taille variable

dont les éléments sont des objets

Yannick.Parchemal@parisdescartes.fr

La classe "java.util.ArrayList"

La classe java.util.ArrayList appartient au paquetage "java.util".

```
/** construit une ArrayList vide (i.e. sans élément) */
public ArrayList( ){…}

/** ajoute 'obj' à la fin de cette ArrayList */
public void add(Object  obj){…}
/** @return l'élément en position 'index' de cette ArrayList */
public Object get(int  index){…}
/** l'élément en position 'index' de cette ArrayList est remplacé par 'obj' */
public void set(int index ,Object obj){…}
/** @return vrai si cette ArrayList contient 'obj' */
public boolean contains(Object obj){…}
/** @return le nombre d'éléments de cette ArrayList */
public int size( ){…}
 /** @return la chaîne représentant cette ArrayList */
public String toString( ){…}
```

Exemple d'utilisation de la classe "ArrayList"

```
package up5.mi.pary.jt.vecteur;
import java.util.ArrayList;

// un programme utilisant la classe ArrayList
public class TestArrayList {
public static void main(String [ ] args)  {
      ArrayList al = new ArrayList();
      al.add ("Essai ");
      al.add ("de la classe");
      al.add ("ArrayList");
      System.out.println("Element d'indice 2: "+ al.get(2));
      System.out. println(al.toString( ));
      al.set(0,"Utilisation");
      System.out. println(al.toString( ));

}}
```

Element d'indice 2: **ArrayList**
[Essai, de la classe, ArrayList]
[Utilisation, de la classe, ArrayList]

Transtypage et ArrayList

```
ArrayList al = new ArrayList();
al.add("BoNjOUr");

String s2=(String)al.get(0);
    // String s2 = al.get(0); serait refusé à la compilation car get rend un Object

System.out.println(s2.toUpperCase());

// Les deux étapes en une instruction (attention au parenthèsage !)
System.out.println(((String) al.get(0)).toUpperCase());
```

Illustration de la liaison dynamique avec la classe ArrayList

```
ArrayList al = new ArrayList();
al.add(new Compte("Dupond",200));
al.add("une chaîne");
al.add(new Date(0));

for (int i=0;i<al.size();i++){
   String s = al.get(i).toString();
   System.out.println(s.toUpperCase());
   }
```

La méthode toString qui est effectivement appelée
n'est pas la même selon les éléments de la liste.
C'est la méthode toString de la classe de l'élément qui est appliquée.
(soit la méthode redéfinie, soit la méthode héritée)

Tableau et ArrayList

String [ ] t; | **ArrayList v;**

Le nombre d'éléments d'un vecteur n'est pas limité.
Le nombre d'éléments d'un tableau est fixé à la compilation

```
t= new String[10];
int nbElt=0;/* le nombre
d'éléments effectif du tableau*/
```

```
v = new ArrayList( );
```

Le nombre d'éléments effectif d'un vecteur est donné par la méthode size.
Il doit être géré explicitement dans le cas de l'utilisation d'un tableau

```
t[nbElt] = "c1";nbElt++;
t[nbElt] = "c2";nbElt ++;
t[nbElt] = "c3";nbElt ++;
……
t[nbElt] = "c";nbElt++;
```

```
v.add("c1");
v.add("c2");
v.add("c3");
……
v.add("c");
```

```
for (int i = 0;i<nbElt;i++)
System.out.println(t[i]);
```

```
for (int i = 0;i<v.size( );i++)
 System.out.println(v.get(i));
```

les classes associées aux types simples

**Il est parfois nécessaire d'utiliser des objets
et non des valeurs de types simples.
C'est pourquoi il existe, pour chacun des 8 types simples,
une classe correspondante.**

Type simple	type classe correspondant
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Yannick.Parchemal@parisdescartes.fr

int et Integer

```
// pour "passer" d'un int à l'Integer correspondant
Integer x = new Integer(6);
System.out.println(x);
// pour passer de l'Integer à l'int correspondant
int ix = x.intValue( );
System.out.println(ix*3);
```

```
6
18
```

```
// un autre constructeur de la classe Integer
Integer y = new Integer("2"+"45");
System.out.println(y);
```

```
245
```

La classe Integer : constructeurs et méthodes d'instances

LES CONSTRUCTEURS

```
/** Constructs a newly allocated Integer object that represents the primitive int argument 'i'*/
public Integer(int i){…}
/** Constructs a newly allocated Integer object
that represents the value represented by the string 's'*/
public Integer(String s){…}
```

LES METHODES D'INSTANCES

```
/** @return the value of this Integer as an int*/
public int intValue(){…}
/** @return the value of this Integer as a double*/
public double doubleValue(){…}
/** @return  the value of this Integer as a float*/
public float floatValue(){…}
/** Compares this Integer to the specified object 'o'*/

public boolean equals(Object o){…}
/** @return a String object representing this Integer's value*/
public String toString(){…}
```

La classe Integer : membres statics

```
int lePlusGrand=Integer.MAX_VALUE;
System.out.println(lePlusGrand);


System.out.println(Integer.toBinaryString(245));

System.out.println(Integer.toHexString(245));

int i = Integer.parseInt("-56");

System.out.println(i);
```

```
2147483647
11110101
f5
-56
```

La classe Integer : membres statics

LES DONNES MEMBRES (CONSTANTES)

/** The largest value of type int */
public static final int MAX_VALUE ;
/** The smallest value of type int */
public static final int MIN_VALUE;

LES METHODES DE CLASSES

/** Parses the string 's' as a signed decimal integer.*/
public static int parseInt(String s){…}

/** @return a string representation of the integer 'i' as an unsigned integer in base 2*/
public static String toBinaryString(int i){…}

/** @return a string representation of the integer 'i' as an unsigned integer in base 16*/
public static String toHexString(int i){…}

/** @return a new String object representing the specified integer 'i'*/
public static String toString(int i){…}

java.util.ArrayList et java.lang.Integer exemples

```
java.util.ArrayList al = new java.util. ArrayList();
// ajout de cinq éléments dans la liste
for (int i=1;i<=5;i++) al.add(new Integer(i));
// affichage de la somme des éléments de la liste
System.out.println(sommeEntiere(al));
// modification des éléments de la liste
for (int i=0;i<al.size();i++){
String s = al.get(i).toString();
al.set(i, new Integer(s+s));}
// calcul de la nouvelle somme des éléments de la liste
System.out.println(sommeEntiere(al));
```

15

165

java.util.ArrayList et java.lang.Integer : somme des éléments d'une liste d'integer

```
/** rend la somme des éléments d'une liste d'Integer  */
public static int sommeEntiere(ArrayList al){
int res = 0;
for (int i=0;i<al.size();i++){
    Integer n = (Integer) al.get(i);
    res+=n.intValue( );
    }
return res;
}
```

Tables de hachage : la classe java.util.HashMap

Les tableaux (ainsi que les listes) permettent
de retrouver directement les éléments connaissant leur indice

Les tables de hachage permettent de retrouver les éléments à partir de **clefs**

```
HashMap h = new HashMap( );
h.put("Lucie","1.43.12.12.32");
h.put("Loïc","1.43.36.48.91");
h.put("Anaïs","2.34.32.64.01");
```

**Dans cet exemple,
les valeurs de la HashMap
sont des numéros de téléphone
Les clefs sont des noms**

```
String nom = term.readString("Nom ?");
System.out.println(h.get(nom));

nom = term.readString("Nom ?");
System.out.println(h.get(nom));
```

```
Nom ? Lucie
1.43.12.12.32
Nom ? Pierre
null
```

Yannick.Parchemal@parisdescartes.fr

Les méthodes get et put de HashMap

public Object get(Object key)

Returns the value to which the specified key is mapped in this hashMap.

Parameters: key - a key in the hashMap.

Returns: the value to which the key is mapped in this hashMap; null if the key is not mapped to any value in this hashMap.

public Object put(Object key,Object value)

Maps the specified key to the specified value in this hashMap.

The value can be retrieved by calling the get method with a key that is equal to the original key.

Parameters: key - the hashMap key.

value - the value.

Returns: the previous value of the specified key in this hashMap, or null if it did not have one.

Classes abstraites

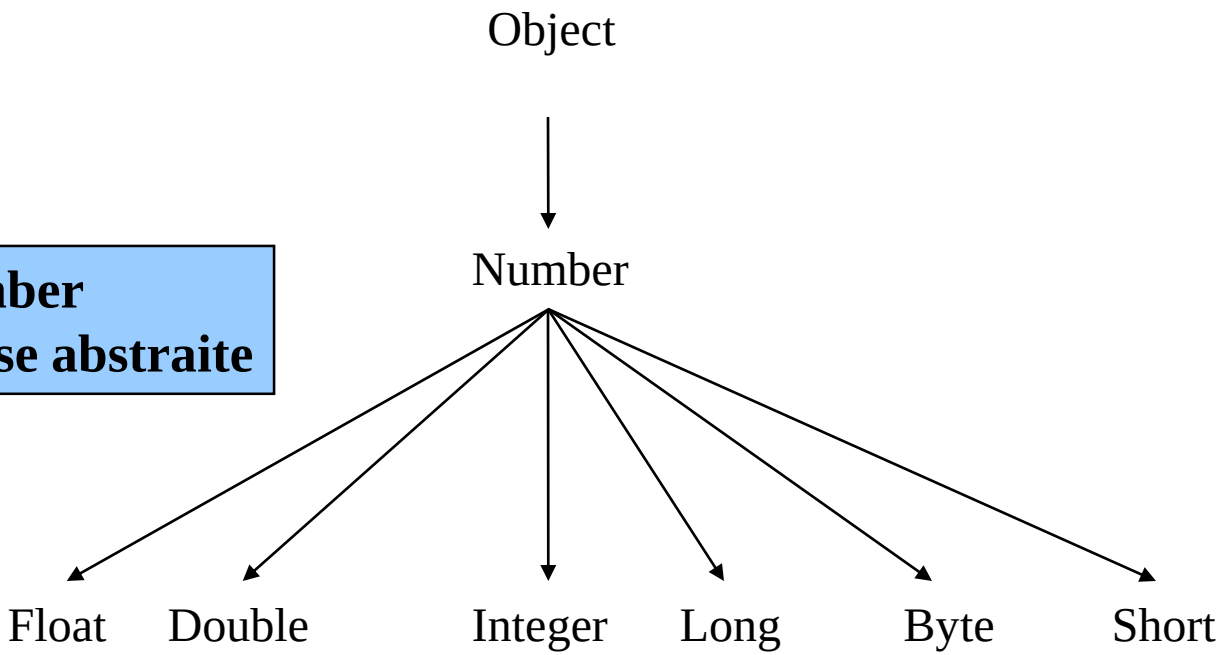
Une classe abstraite est une classe conçue dans l'objectif d'être dérivée.

On ne peut pas créer d'instances directes d'une telle classe.

Certaines méthodes des classes abstraites peuvent être uniquement déclarées (et non définies).
On appelle ces méthodes des méthodes abstraites.

Yannick.Parchemal@parisdescartes.fr

Number, Float, Double, Integer ...



Number
est une classe abstraite

Il n'est pas possible
de créer des instances directes
de la classe Number.

La classe abstraite Number

```
public  abstract class  java.lang.Number  extends  java.lang.Object {

/** rend la conversion en double de ce nombre */
public abstract double doubleValue();
/** rend la conversion en float de ce nombre */
public abstract float floatValue();
/** rend la conversion en int de ce nombre */
public abstract int intValue();
/** rend la conversion en long de ce nombre */
public abstract long longValue();

/** rend la conversion en short de ce nombre */
public short shortValue( ){…}
/** rend la conversion en byte de ce nombre */
public byte byteValue( ){…}
}
```

Pour qu'une sous-classe de Number ne soit pas abstraite, elle doit définir **ces quatre méthodes**.

Les classes abstraites et les méthodes abstraites sont repérées par le mot clé abstract.

Calcul de la somme d'instances de Number

```
public static void main(String [ ] args){
ArrayList al = new ArrayList( );
al.add(new Integer(5));
al.add(new Double(5.3));
al.add(new Short(35));
System.out.println(somme(al));
```

45.3

Integer, Double et Short
sont des instances (indirectes)
de Number

```
/** rend la somme des éléments du vecteur de Number 'v' */
public static double somme(ArrayList al){
double res = 0;
for (int i=0 ; i < al.size( ) ; i++){
    Number n = (Number) al.get(i);
    res+=n.doubleValue( );
    }
return res;
}
```

doubleValue
est une méthode (abstraite)
de la classe Number

Polymorphisme avec doubleValue

```
/** rend la somme des éléments du vecteur de Number 'al' */
public static double somme(ArrayList al){
double res = 0;
for (int i=0;i<al.size( );i++){
    Number n = (Number) al.get(i);
    res+=n.doubleValue( );
    }
return res;
}
```

La méthode doubleValue( ) effectivement utilisée dépend de la classe effective de chaque élément (Integer, Short …).
La classe d'un élément ne peut être Number
(car Number est abstraite)
mais est une sous-classe concrète de Number dans laquelle la méthode doubleValue est (obligatoirement) définie.

Documentation de la classe Integer

```
public  final  class  java.lang.Integer extends  java.lang.Number {
// Fields
    public final static int MAX_VALUE;
    public final static int MIN_VALUE;
// Constructors
    public Integer(int  value){…}
    public Integer(String  s) {…}
// Methods
    public boolean equals(Object  obj) {…
    public int hashCode() {…}
    public String toString() {…}
    public double doubleValue() {…}
    public float floatValue() {…}
    public int intValue() {…}
    public long longValue() {…}
    public static Integer getInteger(String  nm) {…}
    public static int parseInt(String  s) {…}
    public static int parseInt(String  s, int  radix) {…}
    public static String toString(int  i) {…}
    public static String  toString(int  i, int  radix) {…}
    public static Integer valueOf(String  s) {…}
    ....
}
```

définition de deux constantes

les deux constructeurs

quelques méthodes de la classe Object qui sont redéfinies

Les méthodes déclarées dans la classe Number et qui doivent être redéfinies (sinon la classe Integer serait aussi une classe abstraite).

les autres méthodes sont des méthodes de classe.

abstract

méthodes abstraites

Une méthode qui n'est pas définie dans une classe (mais qui est déclarée) est une méthode abstraite. Seule la signature est donnée.

classes abstraites

Si une classe possède une méthode abstraite
soit en déclarant une telle méthode
soit en ne redéfinissant pas une méthode abstraite héritée
Alors c'est une classe abstraite.

Les interfaces

Une interface est
un ensemble de déclaration de méthodes

Et aussi :

- **de constantes statiques**
- **(depuis version 8) de méthodes statiques**
- **(depuis version 8) de méthodes d'instances définies (« default »)**

Interface et classe abstraite

Une interface correspond à une classe abstraite publique avec les restrictions suivantes :

- **Tous les membres sont de visibilité publique**
- **Pas d'attributs excepté des attributs static constants**
- **Les méthodes d'instances sont abstraites sauf celles déclarées « default »**

Yannick.Parchemal@parisdescartes.fr

Exemple d'interface : java.util.Iterator

```
public  interface  java.util.Iterator
{
   /** teste si il reste encore des éléments à énumérer pour cet Iterator */
   public abstract boolean hasNext( );

   /** renvoie l'élément suivant (le premier au premier appel) de cet Iterator */
   public abstract Object next ( ) throws NoSuchElementException ;

 /** supprime le dernier élément (résultat de next( )) rendu par cet Iterator (optionnel)) */
   public abstract void remove ( ) throws NoSuchElementException ;
}
```

Les Iterator permettent de parcourir les éléments d'une structure.

Ils permettent aussi parfois (avec remove) de supprimer le dernier élément retourné par l'itérateur.

Utilisation de l'interface java.util.Iterator

```
public  interface  java.util.Iterator
{  // teste si il reste encore des éléments à énumérer pour cette énumération
   public abstract boolean hasNext( );
   // renvoie l'élément suivant (le premier au premier appel) de cette énumération
   public abstract Object next ( )  throws NoSuchElementException ;
  // supprime le dernier élément (résultat de next()) rendu par cet Iterator (optionnel))
   public abstract void remove ( ) throws NoSuchElementException ;}
```

Les 2 méthodes next et hasNext suffisent pour effectuer des itérations sur une structure

La boucle de parcours ne dépend pas de la structure énumérée.

```
// Impression de tous les éléments d'une arrayList al:
// la méthode "iterator" de ArrayList renvoie une instance indirecte de Iterator
Iterator  it = al.iterator( );
while (it.hasNext( )){
Object elt = it.next( );
System.out.println(elt);
}
```

Utilisation d'un Iterator

// Impression de tous les éléments d'un Iterator 'it'

```
while (it.hasNext( )){
Object elt = it.next( );
System.out.println(elt);
}
```

// tester l'appartenance de l'élément 'eltRecherche' à un Iterator 'it'

```
boolean trouve = false;
while (it. hasNext( ) &&!trouve){
Object elt = it.next ();
trouve = elt.equals(eltRecherche);
}
```

// déterminer le minimum d'un Iterator d'Integer

```
int min=Integer.MAX_VALUE;
while (it. hasNext( )) {
Integer i = (Integer)it.next ( );
 min = Math.min(min, i.intvalue( ));
}
```

Iterator pour la classe ArrayList

```
ArrayList al = new ArrayList();
al.add("Hello");
al.add("World");

// Impression de tous les éléments d'une arrayList 'al' :
Iterator it = al.iterator();
 while (it. hasNext( )){
Object elt = it.next( );
System.out.println(elt);
}
 System.out.println(it.getClass( ).getName());
```

```
Hello
World
java.util.AbstractList$Itr
```

La méthode iterator de la classe ArrayList rend une instance indirecte de Iterator
C'est en fait une instance directe de **java.util.AbstractList$Itr**
qui mémorise le vecteur parcouru et l'indice courant dans le vecteur.

Utilisation de la boucle for avec les Iterator

```
ArrayList al = new ArrayList( );
al.add("Hello");
al.add("World");

// Impression de tous les éléments d'un vecteur v :
for  (Iterator it = al.iterator( ) ; it.hasNext() ; )  {
Object s = it.next( );
System.out.println(s);
            }
```

Hello
World

java.util.HashMap$HashIterator

```
HashMap h = new HashMap( );
h.put("Lucie","1.43.12.12.32");
h.put("Loïc","1.43.36.48.91");
h.put("Anaïs","2.34.32.64.01");

// itérateur des valeurs
Iterator itValues = h.values( ).iterator( );
while (itValues.hasNext( ))
System.out.print(itValues.next( )+" ");
System.out.println();
```

2.34.32.64.01 1.43.36.48.91 1.43.12.12.32

```
// itérateur des clefs
Iterator itKeys = h.keySet( ).iterator( );
while (itKeys . hasNext( ))
System.out.print(itKeys.next( )+" ");
System.out.println();
```

Anaïs Loïc Lucie

```
System.out.println(itKeys.getClass( ).getName( ));
```

java.util.HashMap$HashIterator

Héritage simple, héritage multiple et interface

Héritage simple — Une classe ne peut dériver que d'une classe

Héritage multiple — Une classe peut dériver de plusieurs classes

Pas de véritable héritage multiple avec JAVA
mais une classe peut dériver d'une classe
et implémenter plusieurs interfaces

La notion d'interface permet en particulier
de compenser l'absence d'héritage multiple

implements

extends — Une classe «dérive» d'une super classe

implements — Une classe «implémente» des interfaces

```
// exemple d'en-tête de classes

public class ArrayList extends AbstractList implements Cloneable, List, Serializable

public class StringTokenizer implements Enumeration
```

Une classe qui implémente une interface est forcément abstraite si elle ne définit pas toutes les méthodes abstraites de l'interface

interfaces vides

Certaines interfaces ne contiennent aucun membre.

exemple : java.lang.Cloneable , java.io.Serializable

Elles ne servent que de 'marqueur'

Exemple de java.lang.Cloneable

Si une classe implémente l'interface Cloneable, cela signifie que ses instances peuvent être clonées par la méthode clone de la classe Object

Exemple de java.io.Serializable

Si une classe implémente l'interface Serializable, cela signifie que ses instances sont sérialisables (transformables en un flux d'octets pour des besoins de sauvegarde ou de transmission sur un réseau)

Dérivation d'interfaces

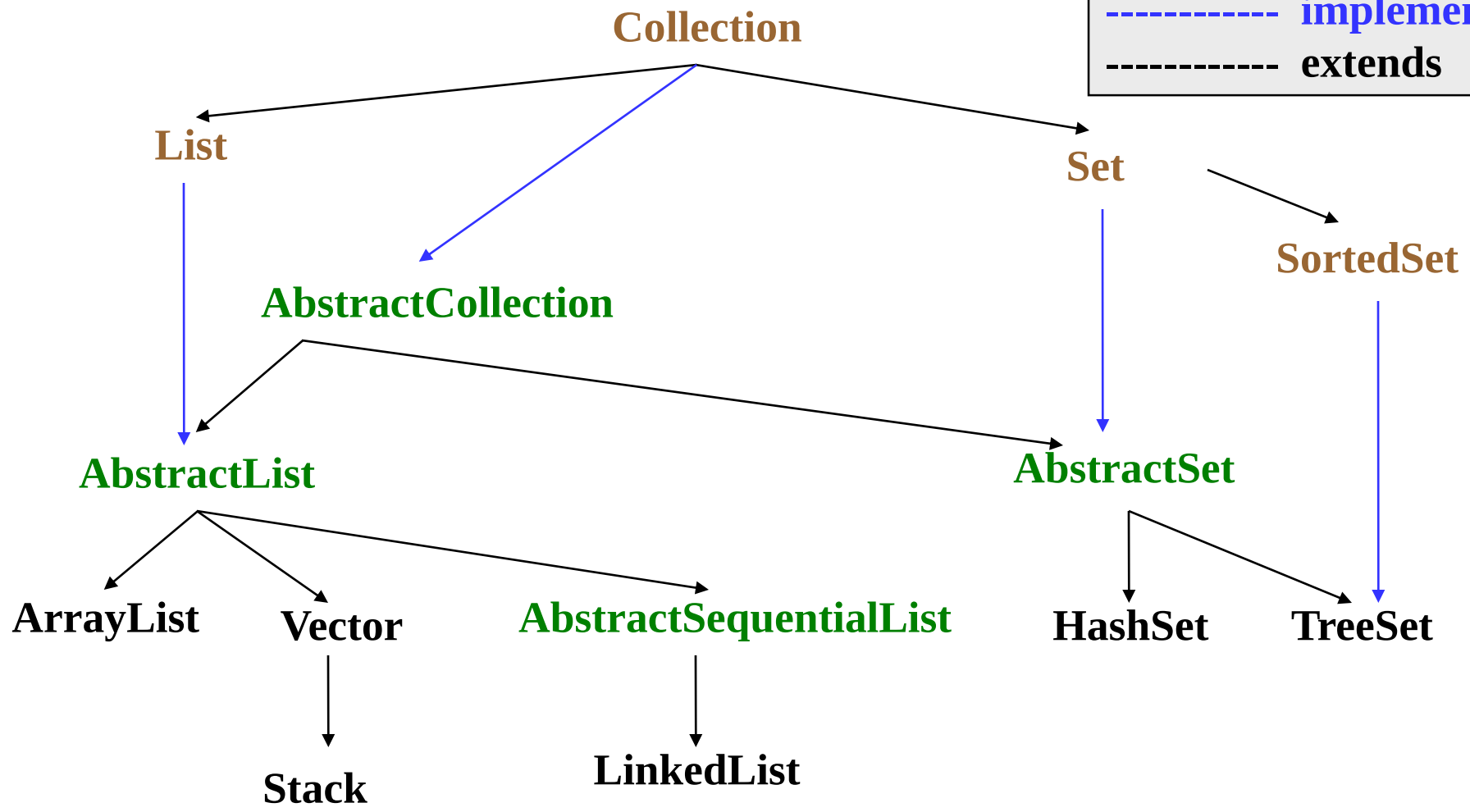
Une interface peut dériver d'une ou de plusieurs interfaces

```
public interface ListIterator extends Iterator{
/** Inserts the specified element into the list (optional operation) */
public void add(Object o) ;
/** Returns true if this list iterator has more elements when traversing the list in the
reverse direction.     */
public boolean hasPrevious( ) ;
/** Returns the index of the element that would be returned by a subsequent call to
next. */
public int nextIndex( ) ;
/** Returns the previous element in the list.  */
public Object previous( ) ;
/** Returns the index of the element that would be returned by a subsequent call to
previous.  */
public int previousIndex( ) ;
/** Replaces the last element returned by next or previous with the specified element
(optional operation). */
public void set(Object o)( );
}
```

Le nombre de membres
de l'interface ListIterator est de 9
- 6 spécifiques
- 3 hérités de l'interface Iterator

Collection

Interface
Classes abstraites
Classes non abstraites
---------- implements
---------- extends



Toutes les classes non abstraites implémentent aussi Serializable et Cloneable

Exemple d'interface : java.util.Collection

```
public  interface  java.util.Collection
{
/** Ensures that this collection contains the specified element (optional operation).*/
public boolean add(Object o);

/**Adds all of the elements in the specified collection to this collection (optional operation)*/
public boolean addAll(Collection c);

public void clear( );
/** Returns true if this collection contains the specified element. */
public boolean contains(Object o);
/** Returns true if this collection contains all of the elements in the specified collection. */
public boolean containsAll(Collection c);

/** Returns true if this collection contains no elements.  */
public boolean isEmpty( );

/** Returns an iterator over the elements in this collection. */
public Iterator iterator( );
```

Exemple d'interface : java.util.Collection

```
/** Removes a single instance of the specified element from this collection, if it is
present (optional operation).
   * @return  true if this collection changed as a result of the call */
public boolean remove(Object o);

public boolean removeAll(Collection c) ;

public boolean retainAll(Collection c) ;

public int size( ) ;

/** Returns an array containing all of the elements in this collection.  */
public Object [ ] toArray( ) ;

…
}
```

Interfaces et liaison dynamique

```
Terminal term = new Terminal("utilisation des interfaces",400,400);
String s = term.readString("Donner le nom d'une classe concrète "+
                                   " qui implémente l'interface Collection\n");

Class cl = Class.forName(s);
Collection c = (Collection) cl.newInstance();

for (int i=0;i<5;i++) c.add(new Integer(i));
if (c.contains(new Integer(3)))term.println("OK");

term.println(c+" "+cl.getName());
```

```
Donner le nom d'une classe concrète  qui implémente l'interface Collection
 java.util.LinkedList
OK
[0, 1, 2, 3, 4] java.util.LinkedList
```

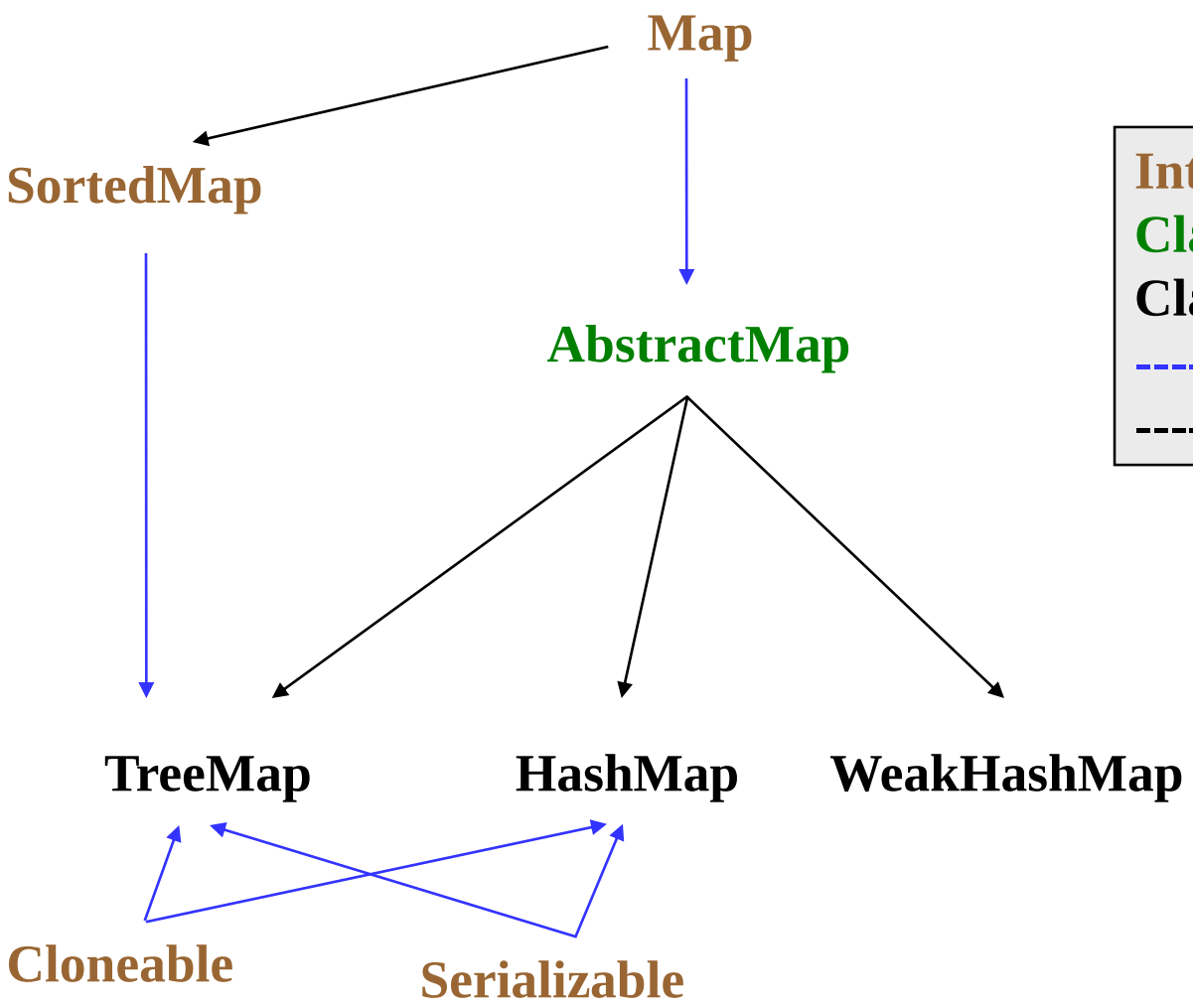
Exemple d'interface : java.util.List

```
public interface List extends Collection
/** Inserts the specified element at the specified position in this list (optional operation). */
public void add(int index, Object element) ;
/** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
public boolean addAll(int index,Collection c);
/** Returns the element at the specified position in this list.
public Object get(int index);
/** Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element. */
public int indexOf(Object o);
/** Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element. */
public int lastIndexOf(Object o);
/** Returns a list iterator of the elements in this list (in proper sequence). */
public ListIterator listIterator( );
```

Exemple d'interface : java.util.List

```
/** Returns a list iterator of the elements in this list (in proper sequence). */
public ListIterator listIterator( );
/** Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list. */
public ListIterator listIterator(int index);
/** Removes the element at the specified position in this list.protected */
public Object remove(int index);
/** Replaces the element at the specified position in this list with the specified element (optional operation). */
public Object set(int index,Object element);
/** Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. */
public List subList(int fromIndex,int toIndex);
}
```

Map



Interface
Classes abstraites
Classes non abstraites
---------- implements
---------- extends

StringTokenizer

Soit une chaîne représentant plusieurs informations séparés entre eux par des caractères séparateurs (espaces, tabulations …)

"30 mars 1999"
"3.5 3.89 23.8 2.3 8.9"
Problème: récupérer ces différents éléments de la chaîne.
La classe StringTokenizer est faite pour cela.

La classe StringTokenizer implémente l'interface Enumeration

```
s=term.readString("donner une date");

for (StringTokenizer st = new StringTokenizer(s); st.hasMoreElements();)
      System.out.println(st.nextElement());
```

```
30 mars 2009
30
mars
2009
```

Exemple d'interface : java.util.Enumeration

```
public  interface  java.util.Enumeration
{
   // teste si il reste encore des éléments à énumérer pour cette énumération
   public abstract boolean hasMoreElements( );
   // renvoie l'élément suivant (le premier au premier appel) de cette énumération
   public abstract Object nextElement( )throws NoSuchElementException ;
}
```

Les Enumeration, comme Iterator, permettent d'énumérer les éléments d'une structure.
(L'interface Enumeration est de moins en moins utilisée au profit de l'interface Iterator)

StringTokenizer

```
/** Etant donnée une chaine représentant une suite de double,
  * rend le ArrayList de Double correspondant
  */
private static ArrayList convertirStringEnArrayListDeDouble(String sDouble){
ArrayList v = new ArrayList( );
try {
    StringTokenizer s = new StringTokenizer(sDouble);
    while (s.hasMoreElements( )){
        Double d = new Double((String)(s.nextElement( )));
        v.add(d);
    }
    }
catch (NumberFormatException e){v.clear();};
return(v);
}
}
```

La classe java.util.StringTokenizer

public class StringTokenizer
extends Object implements Enumeration {

/** Constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is " \t\n\r": the space character, the tab character, the newline character, and the carriage-return character. */
public StringTokenizer(String str) {…}

/** Constructs a string tokenizer for the specified string.*/
public StringTokenizer(String str, String delim) {…}

/** Tests if there are more tokens available from this tokenizer's string. */
public boolean hasMoreElements( ) {…}

/** Returns the next token from this string tokenizer. */
public Object nextElement( ) {…}

...}

Boucles for each : avec des tableaux

```
String [ ] tab = {"dupond","durand"};
for (int i=0;i< tab.length;i++){
String str =tab[i];
System.out.println(str);
      }
```

boucle for classique

```
String [ ] tab = {"dupond","durand"};
for (String str : tab){
    System.out.println(str);
    }
```

boucle for each

for (type var : expression) instruction

expression doit être une collection ou un tableau

Boucles for each avec des listes

boucle for classique

```
ArrayList  list = new ArrayList ( );
list.add("dupond");
list.add("durand");
for (Iterator iterator = list.iterator( ); iterator.hasNext( ); ){
    Object obj=iterator.next( );
     System.out.println(obj);
     }
```

boucle for each

```
ArrayList  list = new ArrayList ( );
list.add("dupond");
list.add("durand");
for (Object obj : list){
      System.out.println(obj);
   }
```

for (type var : expression) instruction

expression doit être une collection ou un tableau

type est le type des éléments de expression

Types génériques
(Classes génériques et interfaces génériques)

Les types génériques existent depuis java 1.5

Exemples de classes génériques

Paramètre formel

```
public class ArrayList<E>

public class HashMap<K,V>
```

Un type générique peut avoir un ou plusieurs paramètres
La valeur de ces paramètres est une classe fixée lors de l'utilisation de ces types

```
// exemple d'utilisation de types génériques
ArrayList <String> al = new ArrayList<String>( );
```

Types génériques : ArrayList

```
ArrayList list = new ArrayList( );
list.add("dupond");
list.add("durand");
for (int i=0;i<list.size();i++){
    String s = (String) list.get(i);
    System.out.println(s.toUpperCase());
    }
```

Version classique

```
ArrayList<String>  list = new ArrayList<String>( );
list.add("dupond");
list.add("durand");
for (int i=0;i<list.size();i++){
    String s = list.get(i);// plus besoin de transtypage
    System.out.println(s.toUpperCase());
    }
```

Version avec type générique

Types génériques : ArrayList

Version classique

```
ArrayList list = new ArrayList( );
list.add("dupond");
list.add("durand");
list.add(new Integer(3));
for (int i=0;i<list.size();i++){
    String s = (String) list.get(i);// erreur à l'exécution
    System.out.println(s.toUpperCase());
    }
```

Version avec type générique

```
ArrayList<String>  list = new ArrayList<String>( );
list.add("dupond");
list.add("durand");
list.add(new Integer(3)); // erreur à la COMPILATION
for (int i=0;i<list.size();i++){
    String s = list.get(i);
    System.out.println(s.toUpperCase());
    }
```

Types génériques : avantages

```
ArrayList<String>  list = new ArrayList<String>( );
list.add("dupond");
list.add("durand");
for (int i=0;i<list.size();i++){
    String s = list.get(i);// plus besoin de transtypage
    System.out.println(s.toUpperCase());
    }
```

2 avantages :
- le paramètre de la méthode add doit être une String (sinon erreur à la compilation)
- plus besoin de transtypage lors de l'appel à get

Types génériques et pseudo code

Les types génériques ne sont utilisés qu'à la compilation.
Le code généré ne les utilisent pas directement.

```
ArrayList <String> al = newArrayList<String>( );
System.out.println(al.getClass( ).getName( ));
```

java.util.ArrayList

Ceci permet la compatibilité avec les versions antérieures

Exemple de la classe ArrayList

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable {

public boolean add(E o) { … }
public E get(int index) { … }
public void clear( ) { … }
}
```

La classe n'est pas dupliquée pour chaque type : une seule version
A la compilation, cela se passe comme si la classe etait dupliquée.

List<String> et List<Object>

```
List<String> listString = new ArrayList<String>( );

List<Object> listObject  = listString ; // ERREUR DE COMPILATION
```

List<String> et List<Object>

```
List<String> listString = new ArrayList<String>( );

List<Object> listObject  = listString ; // ERREUR DE COMPILATION
```

Une liste de string est bien une liste
dont tous les éléments sont des objets
MAIS
une liste d'objets peut contenir autre chose que des strings

listObject.add(new Integer(34));

D'où l'illégalité de l'affectation.

Les wildcards (jokers) <?>

```
List<String> listString = new ArrayList<String>( );

List<Object> listObject  = listString ; // ERREUR DE COMPILATION

List <?> list = listString; // OK !!!

System.out.println(list.get(1)); // compilateur ok

list.add("ert"); // refus du compilateur car il ne connaît pas le type de la liste

String s = list.get(1); // refus

String s = (String) list.get(1); //ok
```

Calcul de la somme d'une liste de Number

```
public static double getSomme (ArrayList list) {
double res=0;
for (Object e : list) {
    res+=((Number)e).doubleValue( );
    }
return res;
}
public static void main(String [] args){
ArrayList list = new ArrayList ( );
list.add(new Double(3));
      list.add(new Double(5));
System.out.println(getSomme(list));
}
```

Sans utilisation des types génériques

Calcul de la somme d'une liste de Number

```
public static double getSomme (ArrayList<Number> list) {
double res=0;
for (Number e : list) {
    res+=e.doubleValue( );
    }
return res;
}
public static void main(String [] args){
ArrayList<Number> list = new ArrayList ( );
list.add(new Double(3));
      list.add(new Double(5));
System.out.println(getSomme(list));
}
```

Avec utilisation des types génériques

Calcul de la somme d'une liste de Number

```
public static double getSomme (ArrayList<Number> list) {
double res=0;
for (Number e : list) {
    res+=e.doubleValue( );
    }
return res;
}
public static void main(String [] args){
ArrayList<Double> list = new ArrayList<Double>( );
list.add(new Double(3));
      list.add(new Double(5));
System.out.println(getSomme(list));
}
```

Problème : le compilateur refuse car ArrayList<Double> n'est pas un sous type de ArrayList<Number>

Utilisation de Jokers (Wildcards)

```
public static double getSomme (ArrayList<? extends Number> list) {
double res=0;
for (Number   e : list) {
    res+=e.doubleValue( );
    }
return res;
}
public static void main(String [] args){
ArrayList<Double> list = new ArrayList<Double>( );
list.add(new Double(3));
      list.add(new Double(5));
System.out.println(getSomme(list));
}
```

Avec l'utilisation du jocker, cela fonctionne.
ArrayList<? extends Number> peut se lire comme ArrayList d'une classe dérivée de Number

Méthodes génériques (1)

```
/** rend le premier element d'une liste different d'un élement donné
    rend null si aucun element n'est different */
public static Object getPremierEltDiff(List list,Object elt){
boolean trouve=false;
int i=0;
while (!trouve && i<list.size())
      if (! list.get(i).equals(elt))
                  trouve=true;
            else i++;
return (trouve)?list.get(i):null;
}
List<String> liste=Arrays.asList(new String[] {"ert","ert","yui","abs"});
String s = (String) getPremierEltDiff(liste, "ert");
```

Méthodes génériques (2)

```
public static String getPremierEltDiff(List<String> list,String elt) {/*meme code*/}
String s = getPremierEltDiff(l1, "ert");

public static <E> E getPremierEltDiff(List<E> list,E elt){/* meme code*/}

String s = getPremierEltDiff(l1, "ert");
```

Les méthodes peuvent aussi être paramétrées.
Le type est déduit au moment de la compilation.

Méthodes génériques : getSomme

```
public static <E extends Number> double getSomme (ArrayList<E> list) {
double res=0;
for (Number   e : list) {
    res+=e.doubleValue( );
    }
return res;
}
public static void main(String [] args){
ArrayList<Double> list = new ArrayList<Double>( );
list.add(new Double(3));
      list.add(new Double(5));
System.out.println(getSomme(list));
}
```

LES EXCEPTIONS

Bloc **try** **catch**(attraper) **finally**

Lancer des exceptions : **throw**

Signaler les exceptions non attrapées: **throws**

Yannick.Parchemal@parisdescartes.fr

ArrayIndexOutOfBoundsException

```
public class TestException{

public static void main(String [] args){
    int[] t = new int[5]; // indices de 0 à 4

    System.out.println(" début ");

    int x = t[5];

     System.out.println("affectation reussie!");

}
}
```

```
début
ERROR: java.lang.ArrayIndexOutOfBoundsException
Press any key to continue...
```

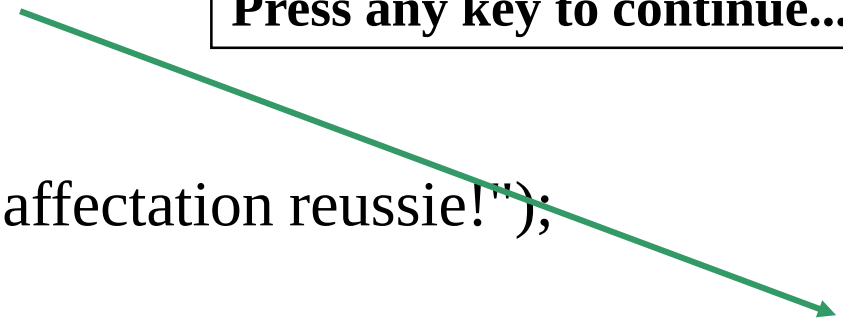
ARRET DU PROGRAMME

Une exception non attrapée provoque l'arrêt du programme

ArithmeticException

```
public class TestException{

public static void main(String[] tArg){
      int[] t = {6,2,7,9,7};

      System.out.println("début");


     int x = 1/(t[4]-t[2]);


      System.out.println("affectation reussie!");

}
}
```

début
ERROR: java.lang.ArithmeticException
Press any key to continue...



ARRET DU PROGRAMME

Ici aussi, l'exception non attrapée provoque l'arrêt du programme

Lancer, attraper une exception

En cas d'anomalie (index en dehors des limites, division par zéro …), une exception est lancée

Une exception lancée et qui n'est pas attrapée provoque l'arrêt du programme avec un message d'erreur

Pour gérer les exceptions,
on utilise des instructions JAVA spécifiques

MOT-CLES liés aux exceptions:
try, catch, finally, throw, throws

Classes d'exceptions

Une exception est une instance d'une classe d'exceptions

La classe java.lang.Throwable est la classe racine des classes d'exceptions.

Quelques classes d'exceptions:

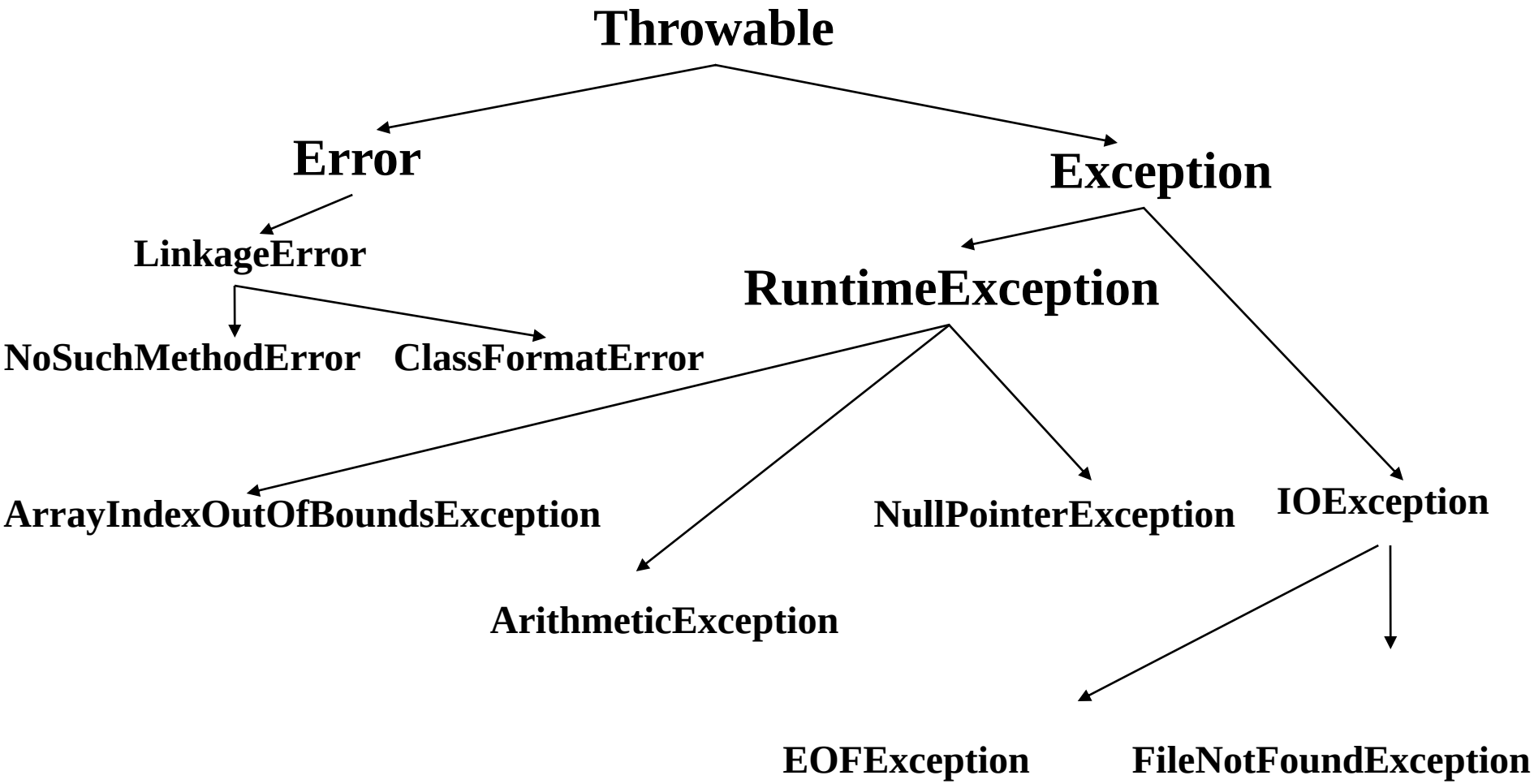
ArithmeticException
pour des erreurs comme la division par zéro

ArrayIndexOutOfBoundsException
indice <0 ou >= au nombre d'éléments du tableau, du vecteur …

NumberFormatException
une chaîne qui n'a pu être convertie en nombre

FileNotFoundException
fichier non trouvé

Quelques classes prédéfinies d'exceptions



catch : pour attraper les exceptions

```
public class TestException{

public static void main(String[] tArg){
    int[] t = {6,2,7,9,7};

    try
        {System.out.println(" début ");
         int x = 1/(t[4]-t[2]);
         System.out.println("affectation reussie! :"+x);
        }
   catch (ArithmeticException e)
         {System.out.println("Operation impossible");}

     System.out.println("FIN");
}}
```

début
Operation impossible
FIN

catch multiples

```
public class TestException{

public static void main(String[] tArg){
   int[] t = {6,2,7,9,7};
     Terminal term = new Terminal("Capture d'erreur");
   try {
      term.println(" debut ");
      int i = term.readInt("Donner un entier ");
      int x = 1/(t[i+2]-t[i]);
      int y = t[i+4];
      term.println("affectations reussies!");
      }

      catch (IndexOutOfBoundsException e)  {term.println("indice incorrect");}
      catch (ArithmeticException e)            {term.println("Operation impossible");}

    term.println("FIN");
    term.end( );
}}
```

```
debut
Donner un entier 4
indice incorrect
FIN
```

```
debut
Donner un entier 0
affectations reussies!
FIN
```

```
debut
Donner un entier 2
Operation impossible
FIN
```

Où capturer les exceptions ?

```
public class TestException{

public static void affecter(int i,int[] tab){
        int x = 1/(tab[i+2]-tab[i]);
        int y = tab[i+4];
    }

public static void main(String[] tArg){
int [ ] t = {6,2,7,9,7};
Terminal term = new Terminal("Capture d'erreur");
try   {     term .println(" debut ");
        int i = term .readInt("Donner un entier ");
        affecter(i,t);
        term .println("affectations reussies!");
    }
     catch (IndexOutOfBoundsException e) {term .println("indice incorrect");}
catch (ArithmeticException e) {term .println("Operation impossible");}
term.println("FIN");
term.end();}}
```

les exceptions
sont souvent capturées
dans les fonctions appelantes

Où capturer les exceptions ?

```
public class TestException{

public static void affecter(Terminal term,int i,int[] tab){
      try {      int x = 1/(tab[i+2]-tab[i]);
                 int y = tab[i+4];}
      catch (ArithmeticException e) {term.println("Operation impossible");}
      term.println("Fin de affecter");
}
public static void main(String[] tArg){
      int [] t = {6,2,7,9,7};
      Terminal term = new Terminal("Capture d'erreur");
      try{term.println(" debut ");
            int i = term.readInt("Donner un entier ");
            affecter(i,t);
            term.println("affectations terminees!");
            }
      catch (IndexOutOfBoundsException e) {
                  term.println("indice incorrect");
            }
      term.println("FIN");
      term.end();}}
```

Le contrôle est transmis au "premier" bloc catch de la bonne classe

```
 debut
Donner un entier 2
Operation impossible
Fin de affecter
affectations terminees!
FIN
```

```
 debut
Donner un entier 4
indice incorrect
FIN
```

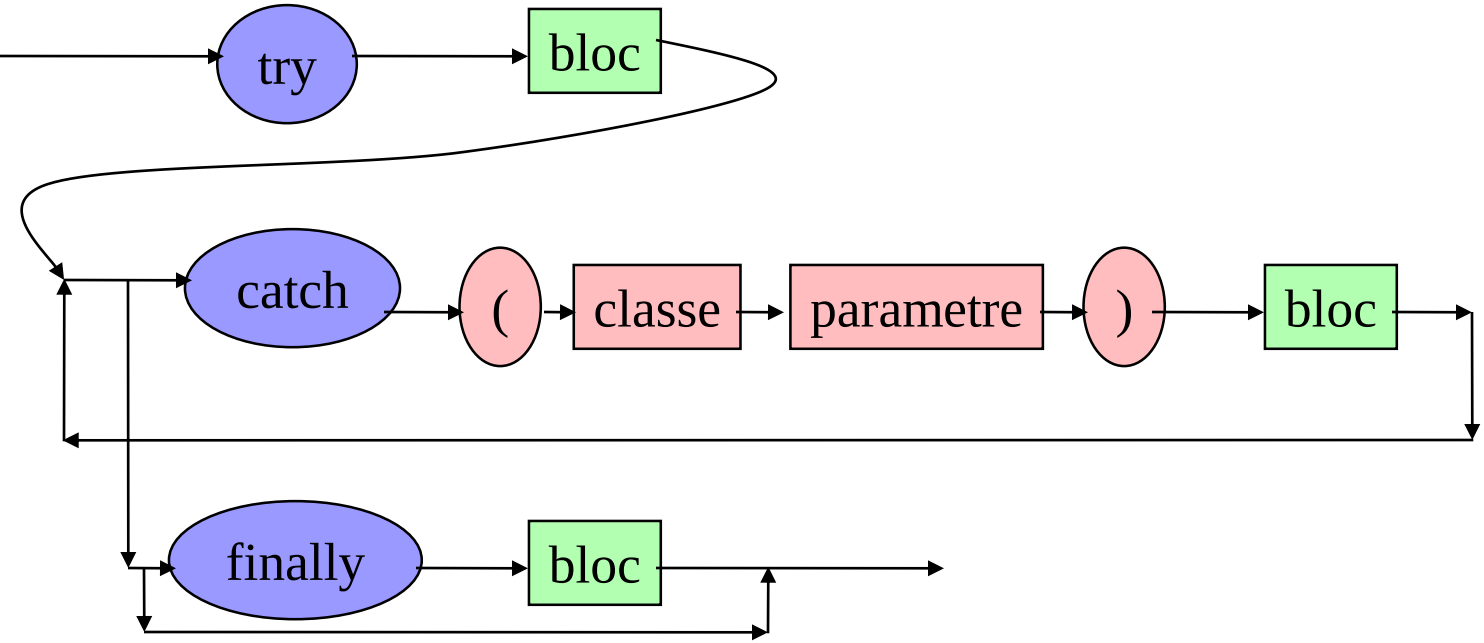
Finally : à éxécuter quoiqu'il advienne

```
ublic class TestException{

ublic static void affecter(Terminal term,int i,int[] tab){
        try {int x = 1/(tab[i+2]-tab[i]);
              int y = tab[i+4];}
        catch (ArithmeticException e) {term.println("Operation impossible");}
        finally {term.println("Fin de affecter");}
    }
ublic static void main(String[] tArg){
    int [ ] t = {6,2,7,9,7};
    try   {term.println(" debut ");
           int i = term.readInt("Donner un entier ");
           affecter(i,t);
       term.println("affectations terminees!");
           }
    catch (IndexOutOfBoundsException e) {
              term.println("indice incorrect");}
    term.println("FIN");
    term.end();}}
```

```
 debut
Donner un entier 2
Operation impossible
Fin de affecter
affectations terminees!
FIN
```

```
 debut
Donner un entier 4
Fin de affecter
indice incorrect
FIN
```

try catch finally : syntaxe



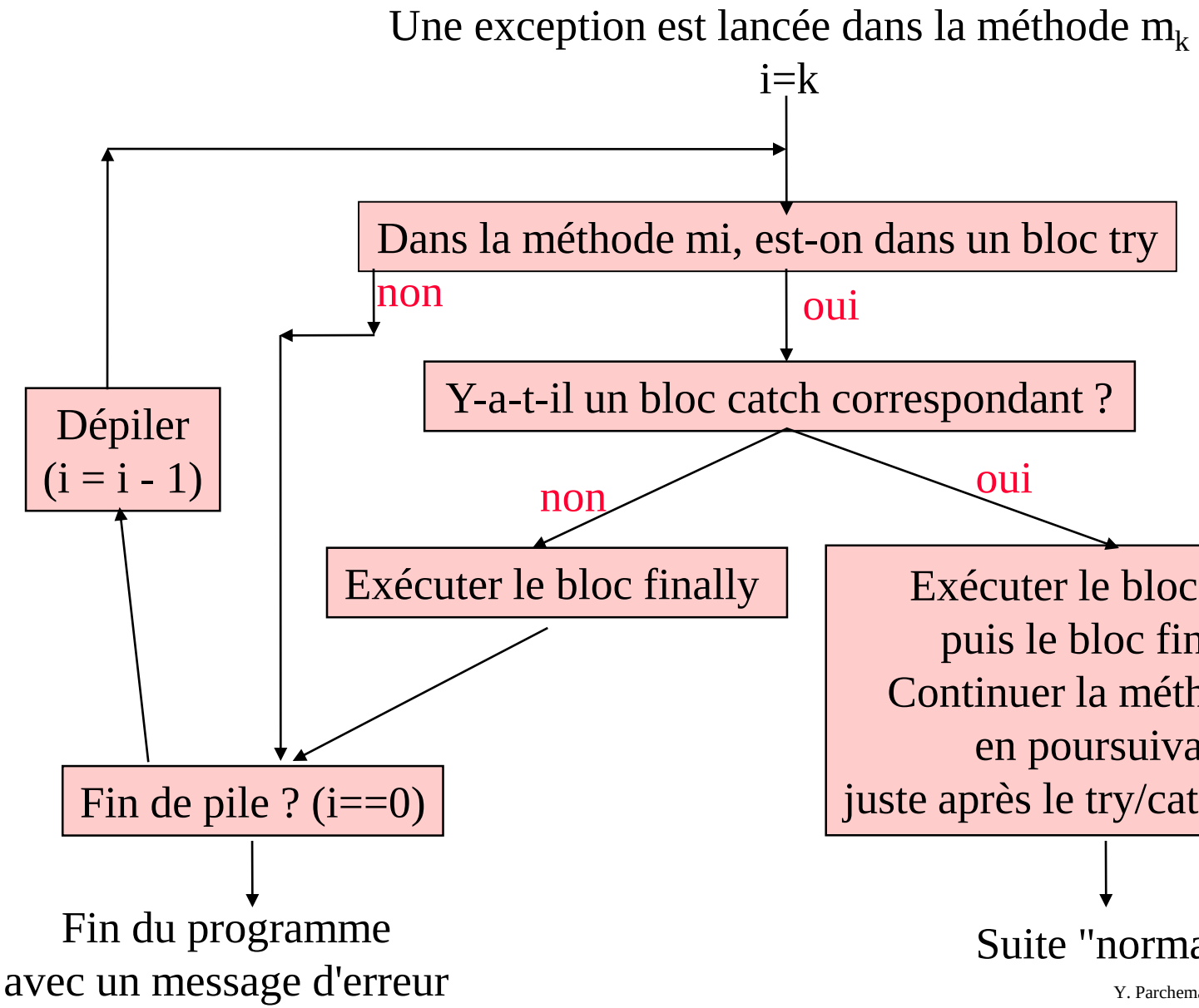
1 bloc try
0, 1 ou plusieurs blocs catch : chacun capture une classe d'exceptions
0 ou 1 bloc finally : exécuté quoiqu'il arrive

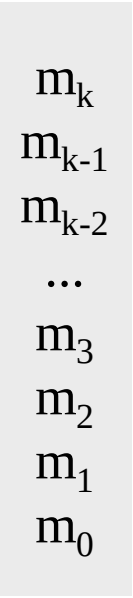
Exemple

```
public class Test {
void f1(int x){
      f2(x);
            System.out.println("f1");}

void f2(int x){
try {
      f3(x);
      System.out.println("f2a");}
      catch (Exception e){
            e.printStackTrace( );
            System.out.println(e);}
            System.out.println("f2b");
}
void f3(int x){
      try {f4(x);}
      finally {System.out.println("final3");}
      System.out.println("f3");
}
void f4(int x){f5(x); System.out.println("f4");}

void f5(int x){int [] t = {5};t[6]=5;}
public static void main(String [ ] args){
Test x = new Test ( );
x.f1(4);
   System.out.println("fin du programme");
}}
```

```
final3
java.lang.ArrayIndexOutOfBoundsException
      at Test.f5
      at Test.f4
      at Test.f3
      at Test.f2
      at Test.f1
      at Test.main
java.lang.ArrayIndexOutOfBoundsException
f2b
f1
fin du programme
```

fonctionnement





La pile

Lancer une exception

Une exception est une instance (directe ou indirecte) de Throwable

Une exception peut être lancée par l'instruction throw <exception>

```
public class Rationnel {
…
public Rationnel (long num,long den){
if (den == 0)
      throw new ArithmeticException("dénominateur nul");
…….
}
```

Gestion des exceptions : throws

Lorsque des exceptions ne sont pas attrapées dans une méthode, on doit le déclarer dans l'en-tête de la méthode avec le mot-clé throws
Syntaxe : throws <ClasseException> [, <ClasseException>]*

```
public void sauvegarder( ) throws java.io.IoException{
    … }
```

les exceptions non traitées doivent obligatoirement être déclarées sauf les instances de :
Error
RuntimeException