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

classes et objets

classes → **objets**

constructeurs —créent→ **identité**

attributs —pour définir→ **état**

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

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

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);
}
```

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

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

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

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

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

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

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

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

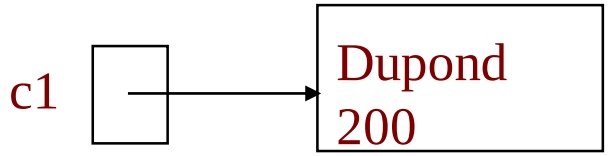
```
i=i+1;
```
i 8

j 7

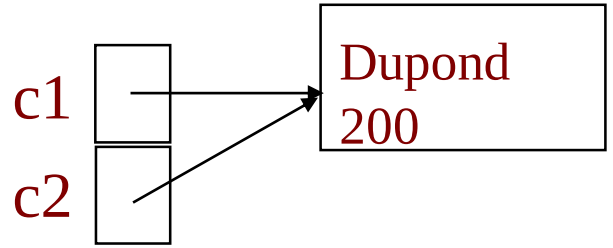
mais les deux variables ne sont pas liées

Affectation sans recopie pour les objets

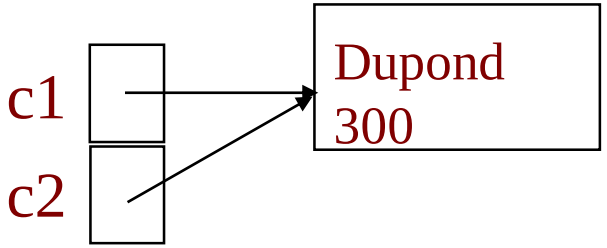
```
Compte c1 = new Compte("Dupond",200);
```




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



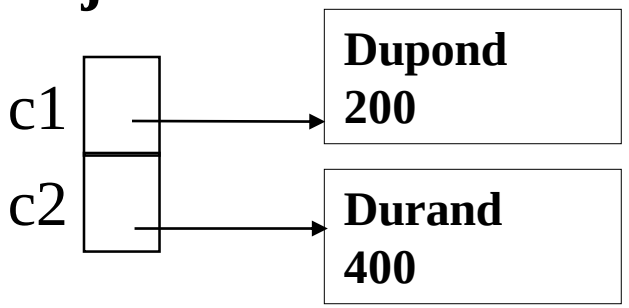

```
c1.addOperation(100);
/* on aurait pu écrire
c2.addOperation(100);
*/
```



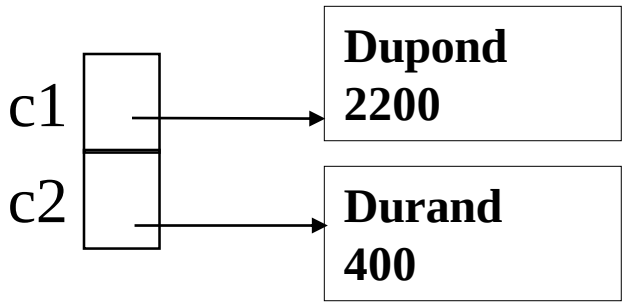
Objets et référence sur des objets

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

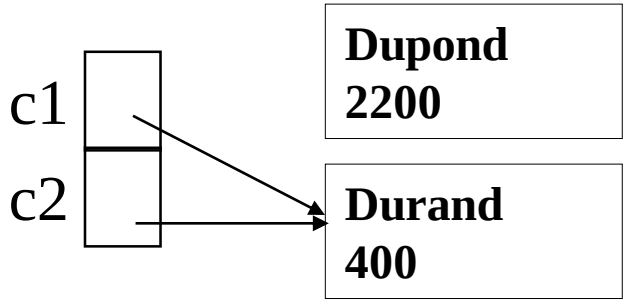
```
Compte c1,c2;
c1 = new Compte("Dupond",200);
c2 = new Compte("Durand",400);
```




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




```
c1=c2;
```

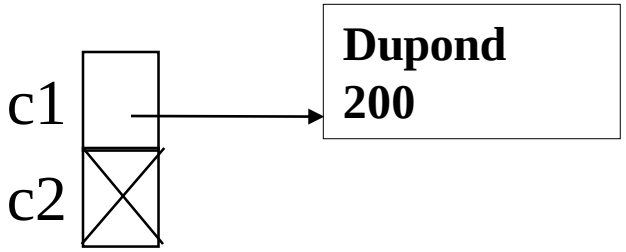


null signifie "absence de référence"

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

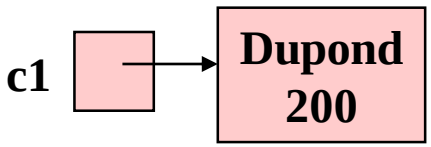
```
public class TestPassage {
private static int f(int j){
      j=j+1;
     return(j);
}
public static void main(String[] tArg) {
      int i = 9;
       System.out.println("i="+i);
      int r = f(i);
      System.out.println("i="+i+" r="+r);
}}
```

i=9
i=9 r=10

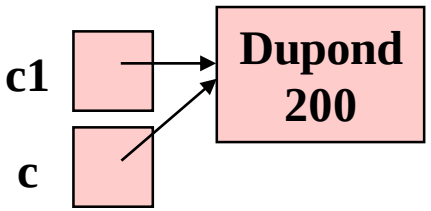
Passage par référence pour les objets

```
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());
}}
```

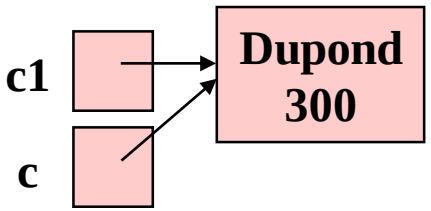
c1:200
c1:300



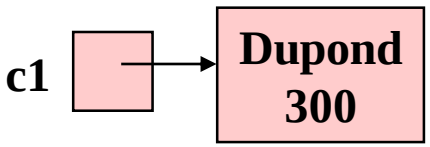
Initialisation de c1



Appel de f



Instruction de f

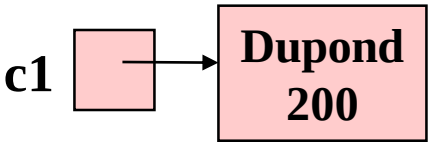


Après le retour de f

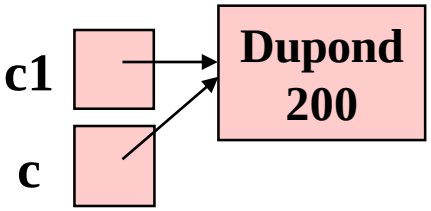
Passage par référence pour les objets

```
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( ));
}}
```

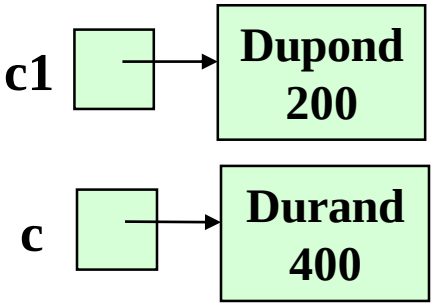
c1:200
c1:200



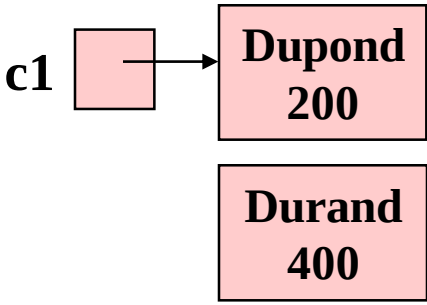
Initialisation de c1



Appel de g



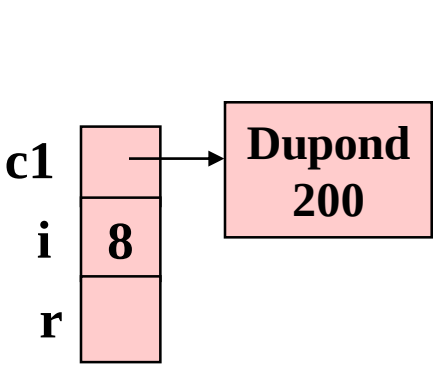
Instruction de g



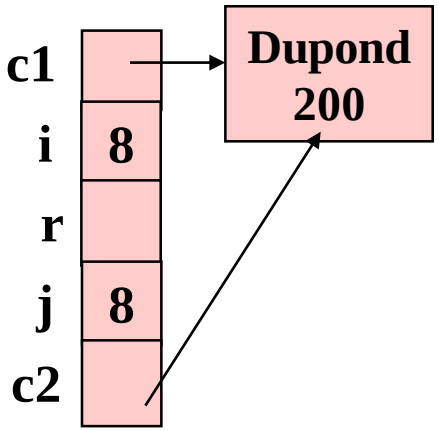
Après le retour de g

Passages par valeur et référence : un autre exemple

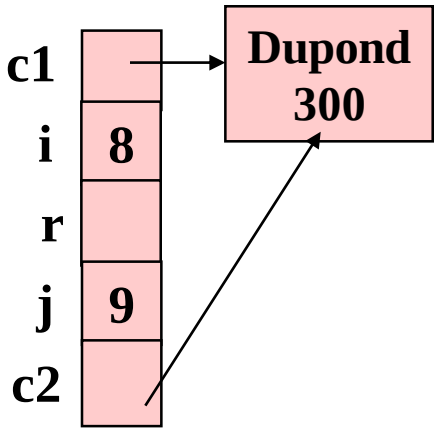
```
public class TestPassage {
public static int h(int j,Compte c2){
        j=j+1;
        c2.addOperation(100);
        return(j);
}
public static void main(String[] args) {
        Compte c1 = new Compte("Dupond",200);
        int i = 8;
        int r = h(i,c1);
        }}
```



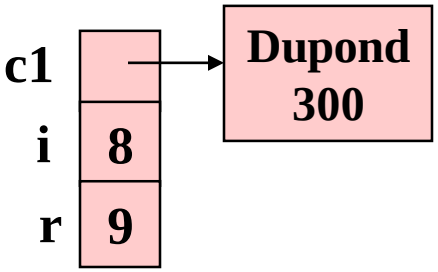
Initialisations



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

Héritage et liaison dynamique

Objet

Classe

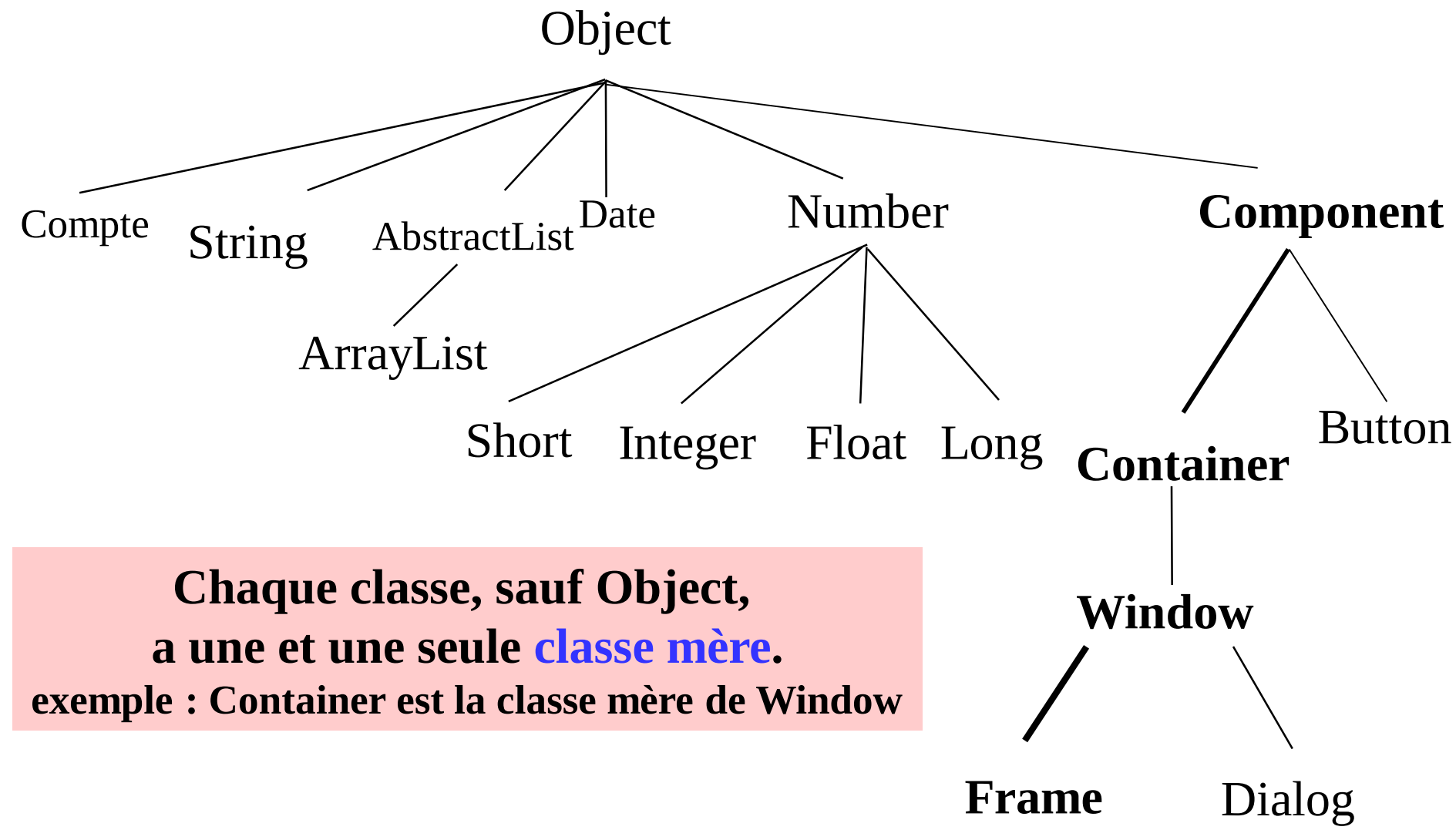
Encapsulation des données

*** 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){…}
```

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
```

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...
```

remarque 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 : 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)

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)

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.

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

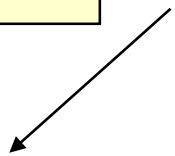
type est le type des éléments de expression

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.