Programmation Java

héritage et récursivité

P. Bakowski

bako@ieee.org

Héritage – classe de départ

Décrivons maintenant la classe permettant de représenter un compte bancaire. Il faut, au minimum, connaître le propriétaire - son nom, le numéro du compte - tableau de caractères, et la somme disponible - double).

Pour les opérations sur un compte, on se limite aux opérations de retrait/virement et la consultation.
Dans le cas où un retrait est impossible on lèvera une exception de type `provisionInsuffisanteErreur` définie séparément.

En plus du constructeur par défaut, nous fournirons egalement un constructeur plus complet permettant de construire un compte en fixant le numéro, le propriétaire et le solde initial.

Nous proposons egalement une méthode permettant d'effectuer un virement d'un compte vers un autre.

Héritage – classe de départ

Décrivons maintenant la classe permettant de représenter un compte bancaire. Il faut, au minimum, connaître le propriétaire - son nom, le numéro du compte - tableau de caractères, et la somme disponible - double).

Pour les opérations sur un compte, on se limite aux opérations de retrait/virement et la consultation.
Dans le cas où un retrait est impossible on lèvera une exception de type `provisionInsuffisanteErreur` définie séparément.

En plus du constructeur par défaut, nous fournirons egalement un constructeur plus complet permettant de construire un compte en fixant le numéro, le propriétaire et le solde initial.

Nous proposons egalement une méthode permettant d'effectuer un virement d'un compte vers un autre.

Héritage – classe de départ

```
public class CompteBancaire {
String nomProprietaire; char[] numero; double solde;
  public CompteBancaire(){}
  public CompteBancaire
          (String proprio, char[] num, double montant){
  this.nomProprietaire=proprio;
  this.numero=num;this.solde=montant;
  }
public double getSoldeCourant(){return this.solde;}
public void deposer(double montant){ this.solde=solde+montant;}
public void retirer(double montant) {
Terminal.ecrireStringln("Retrait sur compte simple");
if(this.solde<montant){
  throw new provisionInsuffisanteErreur;
  } else { solde=solde-montant;}
}
public void virerVers(CompteBancaire c,double montant){
c.retirer(montant); this.deposer(montant); }
}
class provisionInsuffisanteErreur extends Error { }
```

Héritage – classe de départ

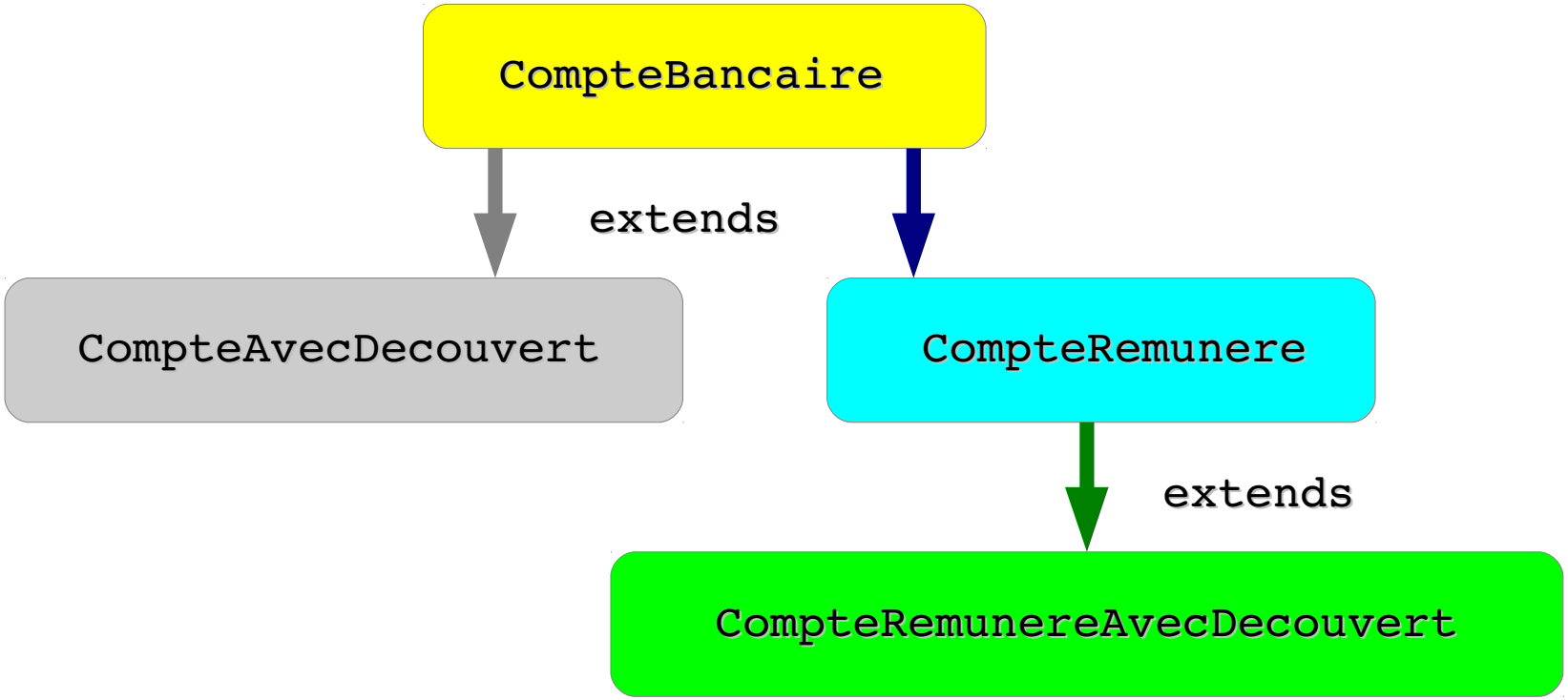
```
class Test1ComptesBancaires {
public static void main(String[] args){
CompteBancaire c1,c2,c3;
String num1 = "123456789";
String num2 = "145775544";
String num3 = "A4545AA54";
c1=new CompteBancaire("Paul",num1.toCharArray(),1000.00);
c2=new CompteBancaire("Paul",num2.toCharArray(),2300.00);
c3=new CompteBancaire("Henri",num3.toCharArray(),5000.00);
c1.deposer(100.00);
c2.virerVers(c1,1000.00);
Terminal.ecrireDouble(c2.getSoldeCourant());
}
}
```

Héritage - extension des données

Supposons maintenant que l'on souhaite prendre en considération la possibilité d'avoir un découvert sur un compte (un retrait conduit a un solde négatif) ou la possibilité d'associer une rémunération des dépôts sur un compte.
Ces cas ne sont pas prévus dans la classe que nous avons définie.
Avec la programmation par objets ceci se réalise naturellement avec la notion d'héritage. Le principe consiste à définir une nouvelle classe a partir d'une classe existante en "ajoutant" des données a cette classe et en réutilisant de façon implicite toutes les données de la classe de base.

Lors de la construction d'une nouvelle classe par extension d'une classe existante, on distingue trois sortes de méthodes (ou de variables) :
- les méthodes (et les variables) qui sont propres à la nouvelle classe ; il s'agit d'extension
- les méthodes (et les variables) qui sont issues de la classe mère ; il s'agit d'héritage
- les méthodes (ou les variables, mais cela est plus rare) qui redéfinissent des méthodes (variables) existantes dans la classe mère ; il s'agit de masquage.

Héritage - extension des données



Extension de CompteBancaire (1)

```
public class CompteAvecDecouvert extends CompteBancaire{
double decouvertMax;
public void fixerDecouvertMaximal(double montant) {
this.decouvertMax=montant ;
}
public CompteAvecDecouvert(String proprio,char[] num,
double montant,double decouvertMax) {
super(proprio,num,montant) ;
this.decouvertMax = decouvertMax ;
}
public void retirer(double montant) {
if (this.solde-montant<-decouvertMax) {
throw new provisionInsuffisanteErreur();
} else { solde= solde-montant;}
}
}
```

Pour la définition du constructeur on utilise le constructeur de la classe mère (par un appel à **super**). Notons que **super** est toujours appelé dans un constructeur soit de manière implicite en début d'exécution du constructeur soit de manière explicite par un appel à **super**.

Extension de CompteBancaire (2)

```
public class CompteRemunere extends CompteBancaire{
double taux; double interets;
public void fixerTaux(double montant) {
this.taux=montant ;
}
public CompteRemunere(String proprio,char[] num,
double montant,double taux) {
super(proprio,num,montant) ;
fixerTaux(montant); interets =0.0 ;
}
public void retirer(double montant) {
if (this.solde<montant) {
throw new provisionInsuffisanteErreur();
} else { solde= solde-montant;}
}
public void calculerInteret() {
interets = interets +1 ; // en realite plux complexe
}
```

Extension de CompteBancaire (3)

```
public class CompteRemunereAvecDecouvert extends
CompteRemunere {
double decouvertMax;
public void fixeDecouvertMaximal(double montant) {
this.decouvertMax=montant ; }
public CompteRemunereAvecDecouvert(String proprio,
       char[] num, double montant, double taux) {
super(proprio,num,montant);
fixeDecouvertMaximal(montant);
}
public void retirer(double montant) {
if (this.solde-montant<-decouvertMax) {
throw new provisionInsuffisanteErreur();
} else { solde= solde-montant;}
}
}
```

Transtypage

La règle liée à la programmation objet est que toute classe est compatible avec ses sous-classes c'est a dire que si `B` est une sous-classe de `A` alors toute variable de type `A` peut etre affectée par une valeur de type `B` (par une instruction d'affectation ou lors d'un passage de paramètre).
Par exemple, si `c1` est un compte bancaire et si `d1` est un compte bancaire avec découvert autorisé alors l'affectation `c1 = d1` est correcte.
Par contre, l'affectation `d1 = c1` sera incorrecte.
De même une instruction `c1.virement(d1,100.0)` sera correcte et il y aura une conversion implicite de `d1` en une valeur de type `compteBancaire` lors de l'appel de la méthode virement.
On peut bien entendu egalement exécuter l'instruction

`d1.virement(c1,100.0)`.

Lors de l'appel `c1.virement(d1,100.0)`, le paramètre formel de la méthode virement désignant le compte sur lequel on va opérer un retrait (nommons le `c`) est associé au paramètre effectif `d1`. Une conversion a lieu.

Liaison dynamique

Dans les exemples précédents, les méthodes appelées à l'exécution du programme sont connues à la compilation.
Supposons maintenant que l'on déclare un compte bancaire `c` sans l'initialiser :

```
CompteBancaire c ;
String num = "AAA4AA54" ;
```

Les règles de typages de Java font qu'il est possible d'associer à `c` par création un `CompteBancaire` ou un `CompteAvecDecouvert`. En effet, toute classe est compatible avec ses sous-classes, et donc, `c` qui est de la classe `CompteBancaire` est compatible avec la classe `CompteAvecDecouvert`.
Les deux instructions ci dessous sont correctes :

```
c=new CompteBancaire("Marie",num.toCharArray(),10000.00);
c=new CompteAvecDecouvert("Marie",num.toCharArray(),10000.00,0.0);
```

Donc, l'instruction `c.retrait(100.00)` appelera soit la méthode retrait des comptes avec découvert (second cas) soit la méthode retrait des comptes ordinaires. Cette liaison (choix) entre appel et méthode peut se faire lors de l'exécution du programme; on parle alors de liaison tardive (dynamique).

Liaison dynamique

```
public class TestLiaisonTardive{
public static void main(String[] args) {
CompteBancaire c ; String num = "AAA4AA54" ;
Terminal.ecrireString("Voulez vous creer un compte decouvert O/N:");
char reponse; reponse = Terminal.lireChar();
if ((reponse!='O')&&(reponse!='o')) {
Terminal.ecrireString("Creation d'un compte ordinaire");
c = new CompteBancaire("Marie",num.toCharArray(),10000.00);}
else {
Terminal.ecrireString("Creation d'un compte avec decouvert");
Terminal.ecrireString("Quel est le decouvert autorise:");
double max;  max = Terminal.lireDouble();
c=new CompteAvecDecouvert("Marie",num.toCharArray(),10000.00,max);
}
System.out.println("solde avant retrait"+c.soldeCourant());
System.out.println("Tentative de retrait de 11000.00");
c.retrait(11000.00);
System.out.println("le nouveau solde="+c.soldeCourant());
}
}
```

Récursivité

Dans certains cas, le sous-problème est une illustration du problème initial, mais pour un cas “plus simple”. Par conséquent, la solution du problème s’exprime par rapport à elle-même !
On appelle ce phénomène récursivité.

Exemple :
Calcul de la factorielle d’une valeur entière positive n (n!=1*2* .... *n)
Mais, n! =(1*2* .... *(n-1))*n, donc n!=(n-1)! *n.
Pour calculer la valeur de n!, il suffit donc de savoir calculer (n-1)! et ensuite de multiplier cette valeur par n. Le sous-problème du calcul de (n-1)! est le même que le problème initial, mais pour un cas “plus simple”, car n−1< n.

```
int factorielle(int n){
int resultat;
if(n==1) resultat=1 ;
else {
int sousresultat = factorielle(n-1); // appel recursif
    resultat = sousresultat*n; }
return resultat;
}
```

Récursivité: test de factorielle()

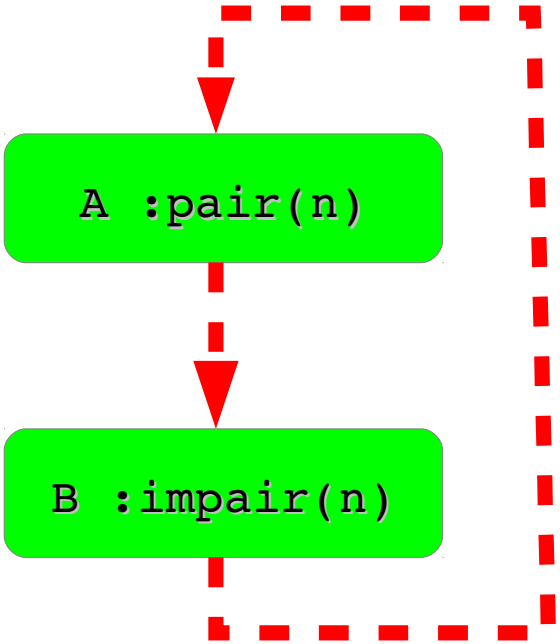
```
public class TestFactorielle{
static int factorielle(int n){
int resultat;
if (n<0) throw new MauvaisParametre();
else if (n==0) resultat=1;
else {
int sousresultat = factorielle(n-1);
resultat = sousresultat * n;}
return resultat;
}
public static void main(String[] args) {
Terminal.ecrireString("Entrez un entier positif:");
int x = Terminal.lireInt();
Terminal.ecrireString(x+"!="+factorielle(x));
}
}
class MauvaisParametre extends Error{}
```

Récursivité directe et indirecte

Quand un sous-programme fait appel à lui même, comme dans le cas de la factorielle, on appelle cela récursivité directe.
Parfois, il est possible qu'un sous-programme A fasse appel à un sous-programme B, qui lui même fait appel au sous-programme A. Donc l'appel qui part de A atteint de nouveau A après etre passé par B. On appelle cela récursivité indirecte (en fait, la suite d'appels qui part de A peut passer par plusieurs autres sous-programmes avant d'arriver de nouveau à A !).

récursivité directe

récursivité indirecte



Récursivité indirecte

```
static boolean pair(int n){
if(n<0)throw new MauvaisParametre();
else if (n==0) return true;
else if (n==1) return false;
else return impair(n-1);
}
```

```
static boolean impair(int n){
if(n<0)throw new MauvaisParametre();
else if (n==0) return false;
else if (n==1) return true;
else return pair(n-1);
}
```

Récursivité indirecte : test

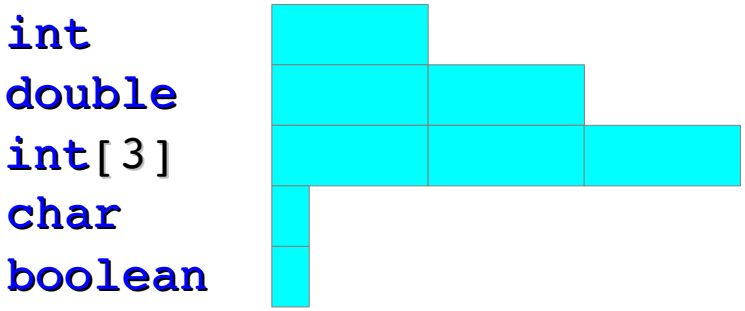
```
public class TestPairImpair {
static boolean pair(int n){
..
}
static boolean impair(int n){
..
}
public static void main(String[] args) {
Terminal.ecrireString("Entrez un entier positif:");
int x = Terminal.lireInt();
if(pair(x))Terminal.ecrireString("nombre pair");
else  Terminal.ecrireString("nombre impair");
}
}
class MauvaisParametre extends Error{}
// fin d'execution
```

Modèle de mémoire

Comment est-il possible d'appeler un sous-programme pendant que celui-ci est en train de s'exécuter ? Comment se fait-il que les données gérées par ces différents appels du même sous-programme ne se "mélangent" pas ?
Pour répondre à ces questions il faut d'abord comprendre le modèle de mémoire dans l'exécution des sous-programmes en Java.
Prenons pour exemple la fonction suivante :

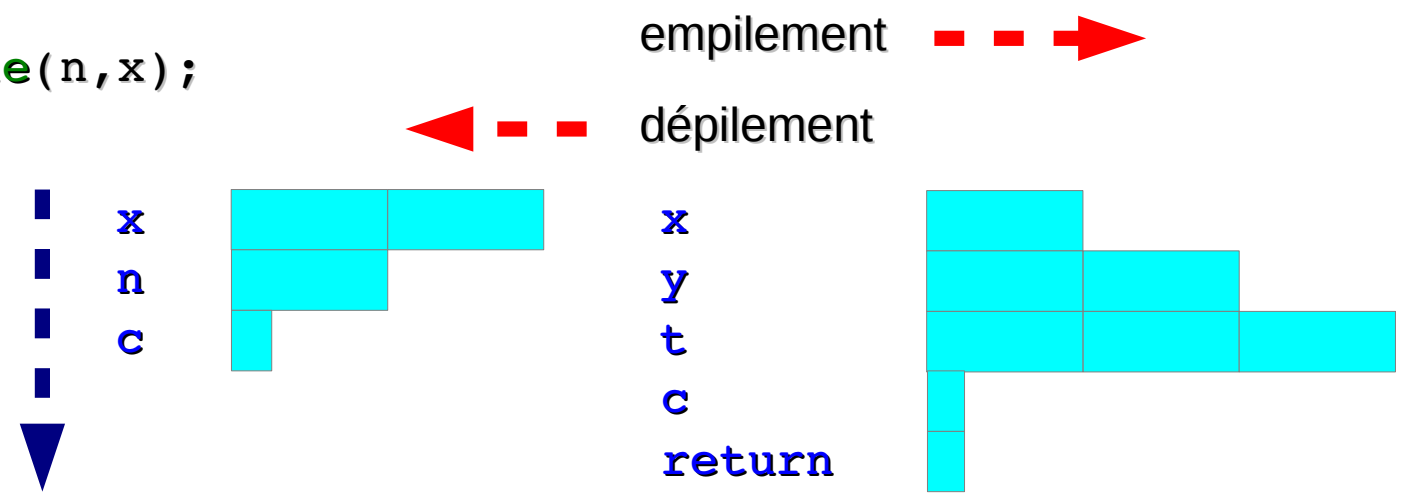
```
boolean exemple(int x,double y) {
int[] t = new int[3];
char c ;
...
}
```

la pile



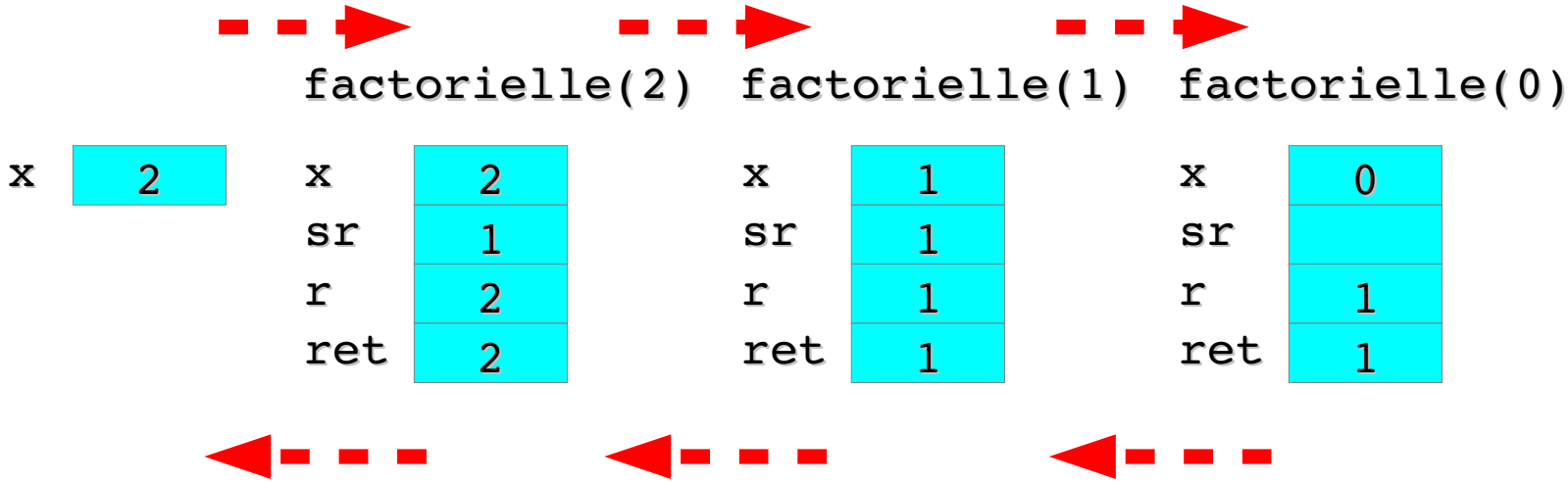
Modèle de mémoire

```
public class Principal {
static boolean exemple(int x, double y ) { . . . }
public static void main(String[] args) {
double x ;
int n ;
boolean c ;
.........
c = exemple(n,x);
.........
}
}
```



Déroulement des appels récursifs

Dans le cas des programmes récursifs, la pile est remplie par l'appel d'un même sous-programme (qui s'appelle soi-même). Prenons le cas du calcul de `factorielle(2)` : le programme principal appelle `factorielle(2)`, qui appelle `factorielle(1)`, qui appelle `factorielle(0)`, qui peut calculer sa valeur de retour sans autre appel récursif. La valeur retournée par `factorielle(0)` permet à `factorielle(1)` de calculer son résultat, ce qui permet à `factorielle(2)` d'en calculer le sien et de le retourner au programme principal.

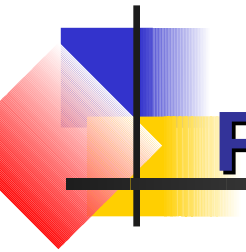


Récursivité et Itération

Suivant le type de problème, la solution s'exprime plus naturellement par récursivité ou par itération. Souvent il est aussi simple d'exprimer les deux types de solution.

```
int nbOccurrences(char c, Strings ) {
int accu = 0 ;
for(int i=0;i<s.length();i ++)
if(s.charAt(i)==c) accu++;
return accu ;
}
```

```
int nbOccurrences(char c, Strings ) {
if (s.length()== 0 ) return 0 ;
// condition d'arret : chaine vide
else {
int sousresultat=nbOccurrences(c,s.substring(1,s.length(-1));
if(s.charAt(i)==c) return 1 + sousresultat;
else return sousresultat;}
}
```



Résumé

Héritage – classe de base

Extension de données (`extends`)

`CompteBancaire` et 2 classes d'extension

Transtypage

Liaisons dynamiques

Récursivité

Récursivité directe et indirecte

Récursivité et Itération