Chapitre 5 : Sous-programmes (2)

(NFA031 - Jour)

V. Aponte

Cnam

7 janvier 2018

Plan du cours

1. Comprendre les sous-programmes : exécution, mémoire
2. Paramètres de type référence.
3. Faire échouer un sous-programme.
4. Surcharge de méthodes en Java.

1. Comprendre les sous-programmes

Appel = Exécution

- Déclarer un sous-programme n'est pas l'exécuter.
- Il faut un appel, pour l'exécuter.
- Chaque appel :
  - produit une exécution nouvelle et indépendante.
  - passe les paramètres nécessaires à la méthode,
  - ex : 2 appels successifs, `valeurAbsolue(-3)`, puis `valeurAbsolue(5)`.

Conclusion : chaque appel

$\Rightarrow$ réalise une exécution nouvelle ;
$\Rightarrow$ passe des paramètres éffectifs différents.

Exécution sous-programmes et mémoire

Exécution $\Rightarrow$ requiert mémoire (stockage variables).

Exécution méthode $\Rightarrow$ utilise mémoire locale avec (paramètres + var.locales)

- inaccessible en dehors du sous-programme ;
- pour chaque appel $\Rightarrow$ nouvelle mémoire ;
- active pendant cette exécution :
- fin d'exécution $\Rightarrow$ mémoire locale « désactivée ».

Retour sur exécution d'un appel

La méthode `main` appelle la méthode `plusUn` :

```
static int plusUn(int x){
  int res = x+1; return res;
}
public static void main (String [] args){
  int x = 3;
  int y = plusUn(x*2); // <--- appel
  Terminal.ecrireStringln("Resultat= "+y);
}
```

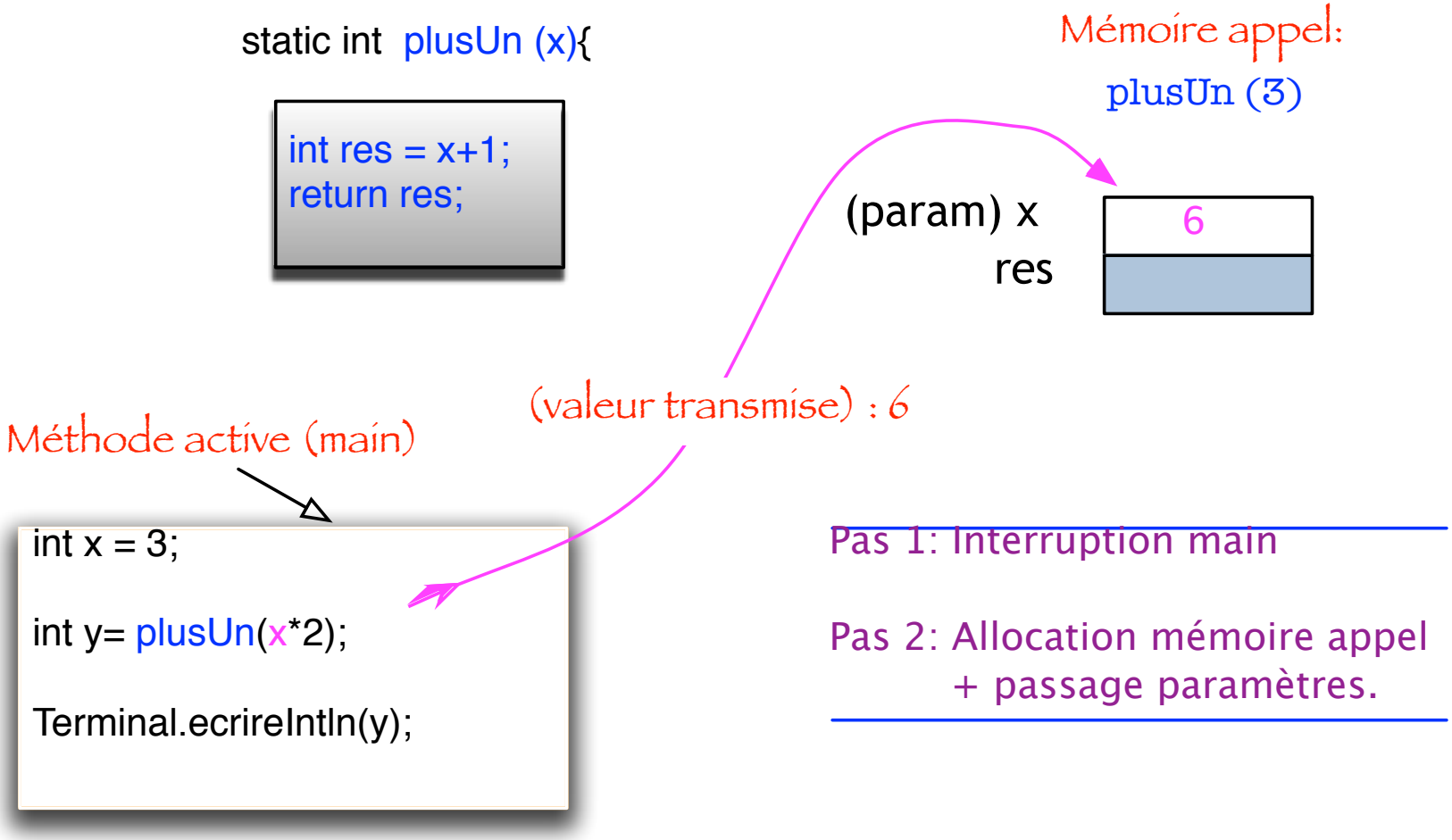
Avec quels arguments se fait l'appel ? ⇒ plusUn(6)

Retour sur exécution d'un appel (2)

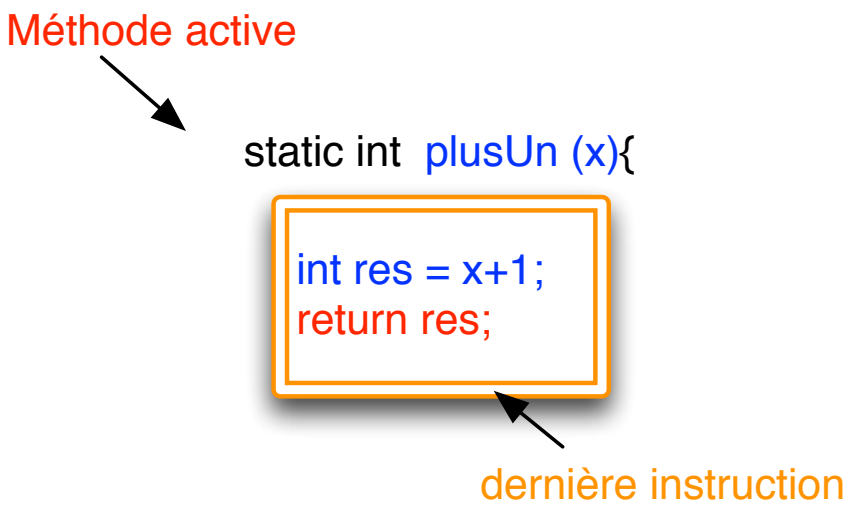
Appel plusUn(6) :

1. Interruption méthode appelante (main) ;
2. Allocation mémoire locale `plusUn` (2 variables) + passage des entrées (paramètres) :
   1. recopie valeurs paramètres : $6 \mapsto x$
3. Exécution `plusUn` ;
4. Fin exécution : des-activation mémoire + retour (avec résultat) vers la méthode appelante (main) ;
5. Reprise exécution méthode appelante (main)

Exécution appel plusUn



Exécution appel plusUn (2)



Mémoire appel:
plusUn (3)

(param) x	6
res	7

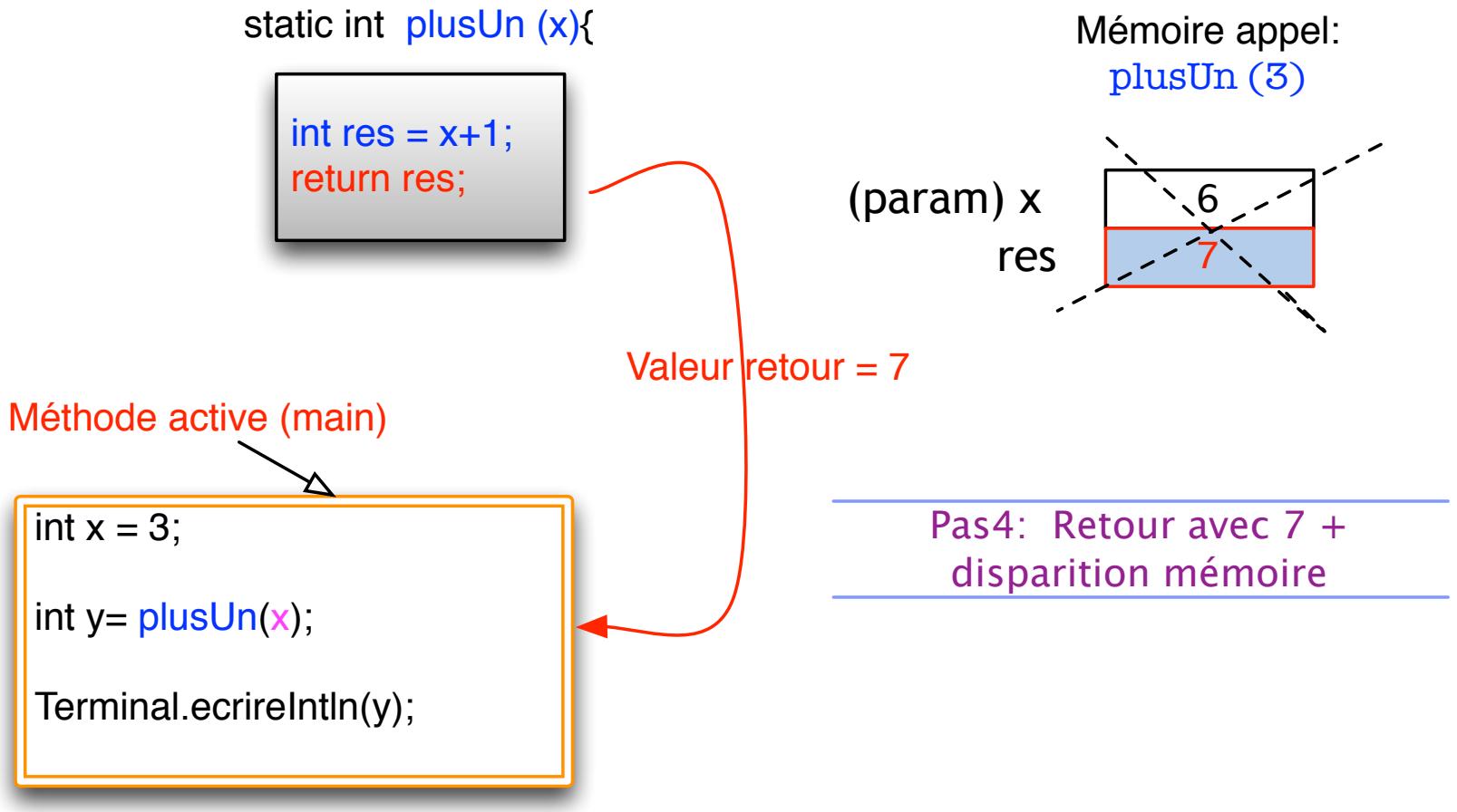
```
int x = 3;

int y= plusUn(x * 2);

Terminal.ecrireIntln(y);
```

Pas 3: Exécution plusUn

Exécution appel plusUn (3)



Exécution appels imbriqués

La méthode `main` appelle la méthode `M` :

```
static int plusUn(int x){
  int res = ....
}
public static void main (String [] args){
  int x = 3;
  int y = plusUn(x); // <--- appel
  Terminal.ecrireStringln("Resultat= "+y);
}
```

- la méthode main possède-t-elle une mémoire locale ?
- quand est-elle active ?
- où est elle passée pendant l'exécution de `plusUn` ?

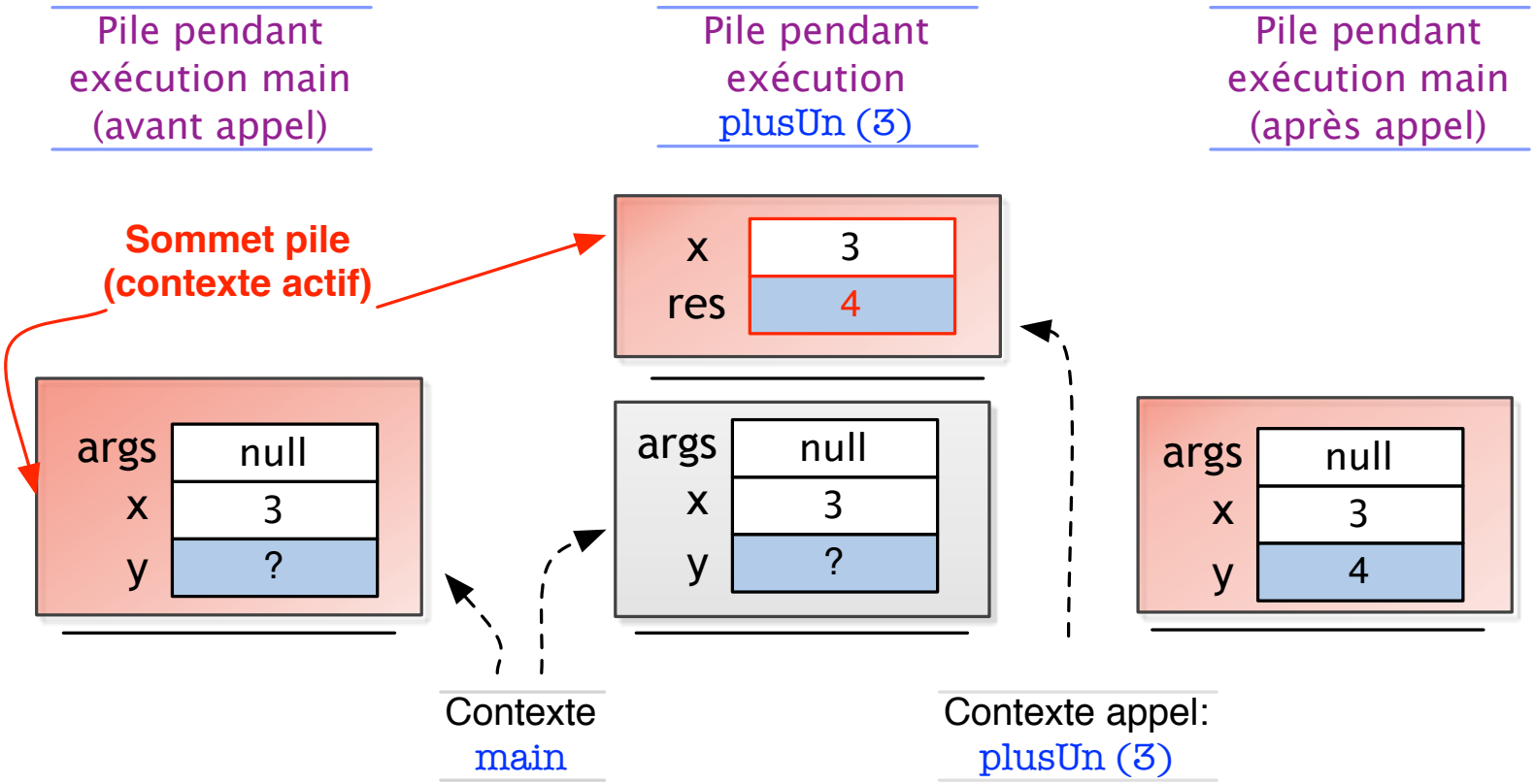
Pile d'exécution

Empilement de mémoires locales correspondant à tous les appels non encore terminés.

- Haut de la pile : mémoire de la méthode active (qui s'exécute) ;
- chaque nouvel appel vers une méthode m, met en place son environnement (mémoire locale) en haut de la pile d'exécution ;
- juste au dessous : mémoire de la méthode appellant m ;
- dès que m est terminée, sa mémoire sort de la pile.
- Se retrouve en haut de la pile $\Rightarrow$ mémoire méthode appelante, qui devient active.

Pile d'exécution (2)

Empilement de mémoires locales correspondant à tous les appels non encore terminés.



2. Paramètres de type référence

De quoi s'agit-il ?

```
static void P(.. n){  // n est local
    ...
}
static void main(..args){
  ... x = v;          // x est local
 P(x);
 // x a pu changer ici?
}
```

- variables+params sous-programme ne sont connus que de lui ;
- enregistrés dans une mémoire locale ;
- elle n'est active que le temps d'exécuter le sous-programme ;
- elle n'est visible que par le sous-programme.

Question : au retour, x peut-il avoir changé ?

Passage de paramètres ⇒ recopie de valeurs

```
static int P(.. n){ ...} // n : variable de P
static void main (...){
  .. x = v; // x : variable du main
   P(x); //  appel --> copie valeur x dans n (memoire P)
}
```

Passage de paramètres

Copie valeurs mémoire appelant ⇒ mémoire locale appelée

- appel `P(x)` ⇒ copie valeur de x dans mémoire de P.

- x de type primitif : on passe sa valeur, entier, booléan, etc.
- x de type référence : on passe sa valeur, qui est une adresse.

Au retour, la valeur de x peut-elle avoir changé ?

Exemple : procédure `inversionTab`

Ce sous-programme inverse les valeurs des cases de son tableau argument.

```
public void inversionTab(int [] t){
   int tampon;
   for(int i=0, j= t.length-1; i < j; i++, j--) {
        tampon = t[i];
        t[i] = t[j];
        t[j] = tampon;
   }
}
```

Procédure `inversionTab` (2)

```
public static void main(String [] args){
   int [] v = {3,7,9, 10};
   inversionTab(v);
   for(int i=0; i < t.length; i++) {
        Terminal.String("␣"+ v[i]);
   }
}
```

- qu'affiche ce programme ?
- le tableau v change-t-il après l'appel ?

Passage avec argument de type référence (tableau)

```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
    int [] x = {1,2,3};
    m(x);
    for (int i=0; i< x.length; i++){
     Terminal.ecrireString(x[i] + "␣");
    }
}
```

Qu'affiche ce programme ?

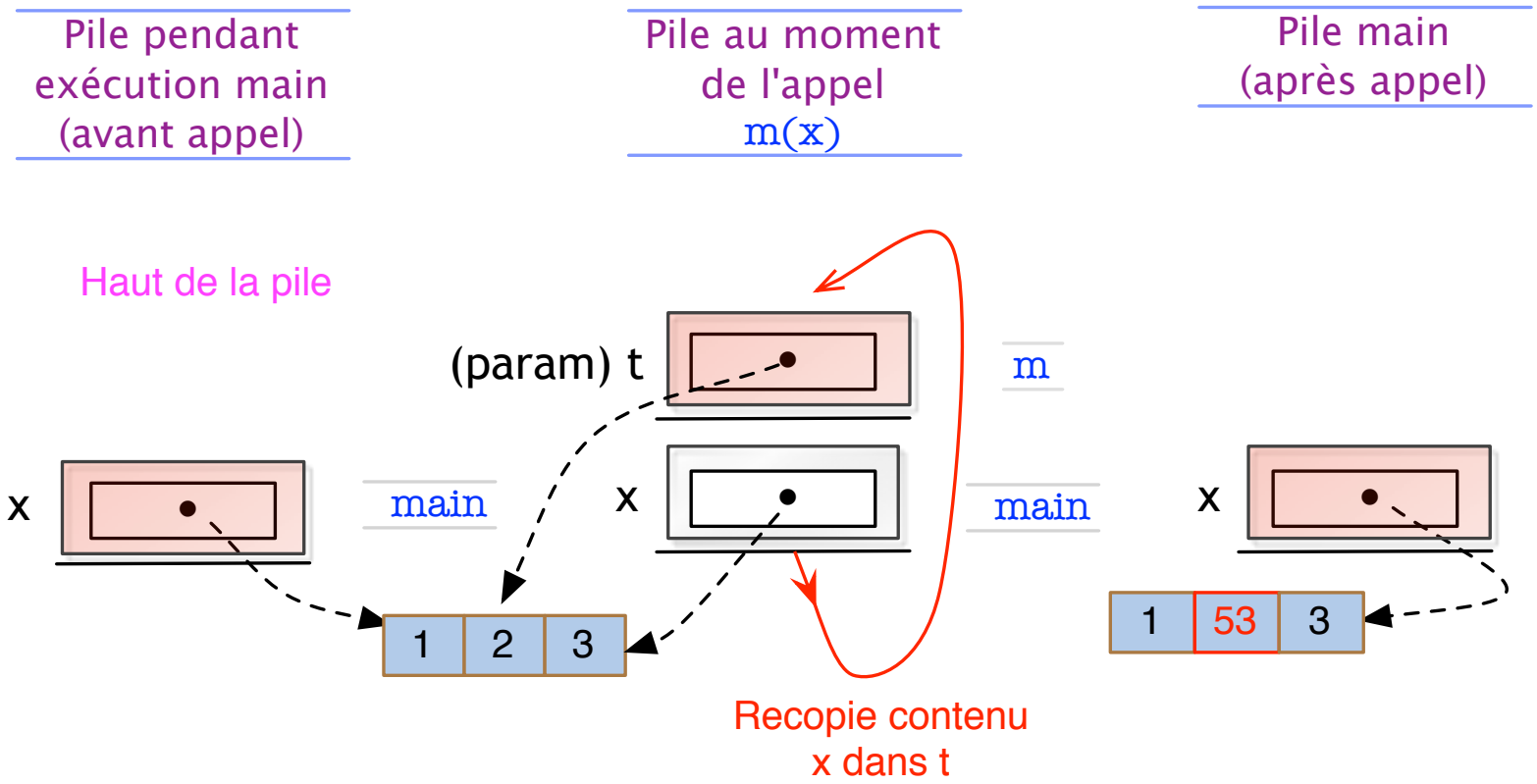
Passage avec type référence (2)

```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
   int [] x = {1,2,3};
   m(x);
   for (int i=0; i< x.length; i++){
    Terminal.ecrireString(x[i] + "␣");
   }
}
```

- le paramètre de `m` est un tableau (type référence) ;
- `p` appelle `m(x)` ⇒
  - x est une variable locale à p,
  - au retour, m a changé la valeur de x ?

Pile d'exécution + paramètres type référence

Pendant l'exécution de m, x et t pointent sur le même tableau ⇒ m peut changer les composantes de x.



Passage de paramètres (bilan)

- Si x est de type référence : on passe sa valeur qui est une adresse.
  - m **ne peut pas changer** l'adresse contenue dans x : `p = truc;` ne change pas x.
  - mais, m **peut changer** une des composantes de la structure se trouvant à cette adresse.
  - Exemples : composante d'un tableau, `p[0] = 17`, variable d'instance d'un objet, `p.jour = 4`.
  - Comme p et x pointent vers le même objet, un changement via p, change indirectement le contenu de x.

- Si x est de type primitif : sa valeur **ne peut pas changer** au cours de l'exécution de m.

Passage de paramètres (suite bilan)

Modifier une variable de type référence *via* un sous-programme :

- définir une fonction qui retourne la nouvelle valeur (adresse)

  ou

- définir une procédure qui *modifie en place* son paramètre :
  - beaucoup de procédures sont définies de la sorte,
  - c'est particulièrement le cas en programmation objet !
  - important pour la suite (2ème semestre)...

Procédure `inversionTab` (fin)

```
public static void main(String [] args){
   int [] v = {3,7,9, 10};
   inversionTab(v);
   for(int i=0; i < t.length; i++) {
        Terminal.String("␣"+ v[i]);
   }
}
```

- qu'affiche ce programme ? `10 9 7 3`
- le tableau v change-t-il après l'appel ? ses composantes sont inversées

3. Échec et exceptions

Échec d'un programme

C'est quoi ? : arrêt de l'exécution d'un programme en raison de conditions anormales survenues pendant l'exécution. Exemples :

- il ne reste plus de mémoire libre pour stocker les données du programme ;
- le programme tente de lire un entier alors qu'on rentre des lettres ;
- le programme tente de lire un ficher qui n'existe pas . . .

```
public static void main (String [] args){
  Terminal.ecrireString("Entrez un nombre: ");
  int x = Terminal.lireInt();
  Terminal.ecrireStringln("Double du nombre: "+ (x*2));
}
```

Si la valeur entrée ne correspond pas à un nombre, quelle exécution ?

Échec (2)

```
 public static void main (String [] args){
  Terminal.ecrireString("Entrez un nombre: ");
  int x= Terminal.lireInt(); // <-- ici
  Terminal.ecrireStringln("Double du nombre: "+ (x*2));
}
```

```
> java Arret
Entrez un nombre: 5f6
Exception in thread "main" TerminalException
        at Arret.main(Arret.java:4)
```

La méthode `Terminal.lireInt()` lève l'exception `TerminalException`.

Les exceptions (1)

C'est quoi ? : un *objet* qui modélise un échec survenu à l'exécution.

- son état interne peut décrire les données de l'échec (en NFA032)
- il existe des exceptions prédéfinies en Java : ex : NullPointerException, ArrayIndexOutOfBoundsException, IOException ;
- on peut définir ses propres exceptions, ex : TerminalException.

Les exceptions (2)

- *Lever* une exception $\Rightarrow$ *sortie d'urgence* de la méthode courante :
  - arrêt de l'exécution de la méthode courante ;
  - l'exception se propage à travers la pile d'appels jusqu'à trouver un *traitement* ;
  - si non traitée, tout le programme échoue ;
- *Traiter* une exception $\Rightarrow$ permet de prendre les mesures adéquates pour tenter de poursuivre l'exécution.

Autre exemple : fonction partielles

Il y a des cas où le résultat d'une fonction n'est pas défini.

Exemple : la fonction qui renvoie l'indice d'une valeur qu'on cherche dans un tableau. Si elle n'est pas trouvée on ne peut pas renvoyer son indice !

```
public static int indiceDe(int v, int []t){
  for (int i=0;i<t.length ; i++) {
    if (t[i]==v){
      return i;
    }
  }
  // que retourner si non trouvé???
}
```

On peut dans ce cas déclencher une erreur. Le sous-programme sera alors arrêté avec une erreur irrécupérable.

Déclarer et lever une exception

Une nouvelle exception (ici, NonTrouve) est définie par *extension* d'une exception existant déjà.

- Déclarer : `class NonTrouve extends RunTimeError{}`
  - Nous définissons `NonTrouve` par extension de `RunTimeException`.
  - `RunTimeException` est une exception prédéfinie.
  - l'extension pour les objets sera étudié en NFA032 (héritage)
- Lever cette exception : `throw new NonTrouve()`

Exemple : déclarer + déclencher

```
class Chercher{
  public static int indiceDe(int v, int []t){
    for (int i=0;i<t.length ; i++) {
      if (t[i]==v){
        return i;
      }
    }
    throw new NonTrouve();  // Déclenchement
  }
  public static void main(String[] argv){
     int [] tab = {10,20,30,40};
     System.out.println("5 est dans la position: "+ indiceD
  }
}
class NonTrouve extends RunTimeError{} // Déclaration
```

Fin exemple

A la compilation, il n'y a pas de problème.

A l'exécution, il se produit une erreur, avec un message :

```
> java Chercher
NonTrouve
at Chercher.indiceDe(Chercher.java:8)
at Chercher.main(Chercher.java:8)
```

En termes Java, on dit qu'une exception a été levée. Le programme s'arrête.

Traiter une exception

Idée : *entourer* du code pouvant léver une exception par une construction de *de rattrapage* qui prévoit instructions à exécuter, selon l'exception levée.

```
try { <code-pouvant-echouer>
      <code-suite-si-tout-va-bien>
} catch (UneException e) {
       <code-traitement>
}
<code-hors-try>
```

Si le code entouré par le try (*code-pouvant-échouer*) ne lève aucune exception, on continue *son exécution normale* avec :

- *<code-suite-si-tout-va-bien>*,
- puis avec *<code-hors-try>*.

Traiter une exception (2)

```
try {   <code-pouvant-echouer>
        <code-suite-si-tout-va-bien>
} catch (UneException e) {
         <code-traitement>
}
<code-hors-try>
```

Si *<code-pouvant-echouer>* lève une exception :

- si elle est de type `UneException` :
  - exécuter *<code-traitement>*,
  - puis continuer la suite du programme : *<code-hors-try>*.

  L'exception a été traitée ⇒ le sous-programme se poursuit normalement.

- si elle n'est pas de type `UneException`, le programme en cours échoue. On saute *<code-traitement>*, *<code-hors-try>*, on dépile et on continue à chercher un traitement en suivant la pile.

Exemple : traiter l'erreur d'un élément non trouvé (1)

Nous devons traiter le cas où l'appel du main pourrait échouer :

- le code de l'appel doit être « protégé » par un `try .. catch`

```
public static void main(String[] args){
    int [] t = {10,20,30,40};
    Terminal.ecrireString("Entrez un entier: ");
    int x = Terminal.lireInt();
    try {
       int n = indiceDe(x,t);
       Terminal.ecrireStringln(x +" trouve au rang "+n);
    }catch(NonTrouve e) {
     Terminal.ecrireStringln( x+ " non trouve.");
    }
}
```

Comment s'exécute ce programme si le nombre lu est 20 ? Si c'est 5 ?

Exemple : traiter l'erreur d'un élément non trouvé (2)

- Si le nombre lu est 20, l'exception n'est pas levée :
  - le code après l'appel (`Terminal.ecrireStringln...`) se poursuit,
  - et le programme se termine normalement.
- Si le nombre lu est 5, l'exception `NonTrouve` est levée :
  - cela arrête l'exécution courante
  - le code après l'appel est abandonné,
  - l'exécution se poursuit avec le code du `catch`.

Exemple : échec et méthodes imbriquées (1)

```
    public static void p1(int x){
        p2(x+1);
        System.out.println("fin␣p1:␣"+x);
    }
    public static void p2(int y){
        p3();
        System.out.println("fin␣p2:␣"+ y);
    }
    public static void p3(){ throw new RuntimeException();}

    public static void main (String [] args){
      p1(3);
      System.out.println("fin␣main␣");
}
```

Quels sont les affichages de ce programme ?

Exemple : échec et méthodes imbriquées (1)

Si l'échec intervient sur une méthode après des appels en cascade ?

- Les appels ici : main $\rightarrow$ p1 $\rightarrow$ p2 $\rightarrow$ p3
- p3 lève une exception ;
- l'exception se propage à travers les « retours » successifs :
  p3 $\rightarrow$ p2 $\rightarrow$ p1 $\rightarrow$ main
- faisant arrêter toutes ces méthodes, autrement dit : aucune ne peut exécuter l'affichage final ni se terminer normalement.
- Affichages : aucun !

Seule manière de stopper les échecs en cascade : trouver le bon try-catch en chemin !

Suite : traitement dans p2

```
    public static void p1(int x){
       p2(x+1);  System.out.println("fin p1: "+x);
    }
    public static void p2(int y){
       try {
          p3();   // <-- leve exception
          System.out.println("fin p2: "+ y);
       catch (RuntimeException e) {
           System.out.println("recuperation p2: ");
      }
    }
    public static void p3(){ throw new RuntimeException(); }

    public static void main (String [] args){
      p1(3); System.out.println("fin main ");
}
```

Afficihages ?

Affichages pour traitement dans p2

- Les appels : main $\rightarrow$ p1 $\rightarrow$ p2 $\rightarrow$ p3 et p3 lève une exception ;
- l'exception se propage sur les retour p3 $\rightarrow$ p2
- dans p2 : on trouve le bon try-catch :
  - on poursuit l'exécution du code dans le catch trouvé
  - on affiche « récupération p2 »
  - p2 se termine normalement
- on poursuit les retours, mais cette fois normalement, sans propager d'exception :
  p2 $\rightarrow$ p1 $\rightarrow$ main
- chacune de ces méthodes poursuit là où elle était restée, et affiche donc son message final

Affichages :

```
recuperation p2
fin p1 : 3
fin main
```

Traitement dans main

Exercice : quels affichages pour cette version ?

```
    public static void p1(int x){
       p2(x+1);  System.out.println("fin p1: "+x);
    }
    public static void p2(int y){
       p3(); System.out.println("fin p2: "+y);
    }
    public static void p3(){ throw new RuntimeException(); }

    public static void main (String [] args){
     try { p1(3);   System.out.println("fin main ");
     } catch (Exception e){
         System.out.println("fin avec recuperation main ");
     }
}
```

Très utile : récuperer les erreurs de saisie

```
public static void main(String [] args) {
  int x; boolean ok=false;
  while (!ok) {
    Terminal.ecrireStringln("Entrez un entier");
    try {
        x = Terminal.lireInt();
        ok=true;
    } catch (TerminalException e){
        Terminal.ecrireStringln("Erreur de saisie. Recomenc
    }
  }
   Terminal.ecrireStringln("Valeur de x= "+x);
}
```

Comment s'exécute ce programme ?

Méthode de saisie avec récuperation

```
static int saisieInt(String message){
 int res; boolean ok=false;
 while (!ok) {
   Terminal.ecrireStringln(message);
   try { res = Terminal.lireInt();
         ok=true;
   } catch (TerminalException e){
       Terminal.ecrireStringln
           ("Erreur de saisie. Recomencez");
   }
 }
 return res;
}
```

La méthode renvoie l'entier saisit en résultat.

Méthode de saisie avec récuperation (2)

```
public static void main(String [] args) {
   int taille = saisieInt("Taille du tableau?");
   int [] t = new int [taille];
   for (int i=0; i<t.length; i++){
     t[i] = saisieInt("Element "+ (i+1)+ "? ");
   }
}
```

On peut employer la méthode plusieurs fois, par exemple, pour saisir de manière sécurisée, la taille d'un tableau et ensuite, les éléments du tableau.

Comment s'exécute ce programme ?

4. La surcharge

Signature d'une méthode

Pour invoquer une méthode correctement on doit connaître :

- le nom de la méthode,
- le nombre et type de chacun de ses paramètres (donnés dans le même ordre que dans leur définition)

Il n'est pas nécessaire de connaître le nom des paramètres formels.

Signature d'une méthode

Ces informations (nom 'une méthode et type et ordre des arguments) constituent la signature d'une méthode.

C'est tout ce qui est nécessaire pour faire un appel correct.

Exemples de signatures :

```
main(String [] )
valeurAbsolue(int)
ecrireInt(int)
liredouble(double)
```

La surcharge

```
// Cherche un entier dans un tableau d'entiers
 static boolean cherche(int x, int [] t){
   for (int i=0; i<t.length; i++){
          if (x == t[i]) return true;
   }return false;}

// Cherche un caractere dans un tableau de caracteres
 static boolean cherche(char x, char [] t){
   for (int i = t.length-1; i >=0;  i--){
          if (x == t[i]) return true;
   }return false;}

public static void main(String[] args) {
  int [] ti = {1,3, 7};
  char [] tc = {'e', 'f', 'k', 'r'};
  boolean a = cherche('a', tc);
  boolean b = cherche(3, ti);
```

La surcharge

```
public static void main(String[] args) {
   int [] ti = {1,3, 7};
   char [] tc = {'e', 'f', 'k', 'r'};
   boolean a = cherche('a', tc);
   boolean b = cherche(3, ti);
```

- **Deux sous-programme** `cherche` : un pour les tableaux de caractères, l'autre pour les tableaux d'entiers.
- Ils ont le même nom, mais accomplissent des tâches différentes : **sémantique différente**.
- La seule chose qui les distingue est **leur signature**.
- Comment le compilateur sait lequel exécuter lors d'un appel ?

La surcharge

- **Deux sous-programme** `cherche` : on dit que cette méthode est **surchargée**.
- La surcharge est autorisée si toutes les signatures des méthodes **avec même nom** sont distinctes, autrement dit, si les types des **paramètres formels** sont différents pour chaque version de la méthode surchargée.
- $\Rightarrow$ On peut donc determiner laquelle exécuter en examinant le type des arguments **effectifs**.

Exemple de surcharge

```
static boolean cherche(int x, int [] t){ ...

static boolean cherche(char x, char [] t){ ...
```

La surcharge pour les deux méthodes `cherche` est autorisée car leurs signatures sont distinctes :

```
cherche(int, int [])

cherche(char, char [])
```

Exemple de surcharge

```
public static void main(String[] args) {
  int [] ti = {1,3, 7};
  char [] tc = {'e', 'f', 'k', 'r'};
  boolean a = cherche('a', tc);
  boolean b = cherche(3, ti);
```

Pour savoir laquelle des 2 méthodes exécuter, le compilateur examine le type des paramètres éffectifs dans chaque appel :

- appel `cherche('a', tc)`, on utilise la méthode de signature `cherche(char, char [])`.
- appel `cherche(3, ti)`, on utilise la méthode de signature `cherche(int, int [])`.