Chapitre 5 : Sous-programmes

(NFA031 - Jour)

V. Aponte

Cnam

1er décembre 2015

Plan du cours

1. Utilité des sous-programmes.
2. Déclarer, utiliser : procédures, fonctions.
3. Exemples.
4. Comprendre les sous-programmes : exécution, mémoire, passage de paramètres.
5. Faire échouer un sous-programme.
6. Surcharge de méthodes en Java.

1. Utilité des sous-programmes

Les sous-programmes sont indispensables

- entité syntaxique unique pour capturer plusieurs instructions ;
  - plus de recopie de code $\Rightarrow$ on invoque le sous-programme ;
  - programmes plus courts, concis, clairs,
  - modifications du code $\Rightarrow$ concentrés en un seul lieu ;
  - circonscrit changements/erreurs.
- essentiels pour découper/structurer les programmes ;
  - sous-programme devient "super-instruction",
  - programme main $\Rightarrow$ construit par appels à sous-programmes ;
  - aide à se concenter sur ce que le programme doit et non pas au "comment”.

Moyenne des notes dans un tableau

Problème : lire un tableau de notes, calculer et afficher la moyenne.

1. Lire un nombre de notes strictement positif (boucle).
2. Créer un tableau de cette taille et l'initialiser avec autant de notes lues.
3. Afficher l'ensemble de notes.
4. Calculer et afficher la moyenne.

Chacune de ces sous-tâches comporte plusieurs instructions et au moins une boucle.

Reprenons un exemple avec sous-programmes

Problème : lire un tableau de notes, calculer et afficher la moyenne.

```
public static void main(....) {

    int  nbe = lirePos();

    double [] notes = lireTab(nbe);

    Terminal.ecrireStringln("Notes obtenues: ");

    afficheTab(notes);

    Terminal.ecrireString("Moyenne:  " );

    Terminal.ecrireDoubleln(moyenneTab(notes));

}
```

Moyenne des notes dans tableau (2)

Problème : lire un tableau de notes, calculer et afficher la moyenne.

```
public static void main(....) {

    int  nbe = lirePos();

    double [] notes = lireTab(nbe);

    Terminal.ecrireStringln("Notes obtenues: ");

    afficheTab(notes);

    Terminal.ecrireString("Moyenne:  " );

    Terminal.ecrireDoubleln(moyenneTab(notes));

}
```

Moyenne des notes dans un tableau(3)

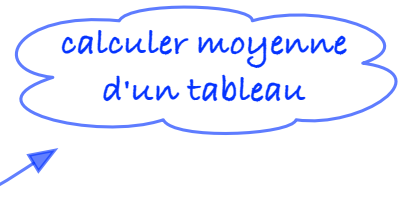
Problème : lire un tableau de notes, calculer et afficher la moyenne.

```
public static void main(....) {

    int  nbe = lirePos();

    double [] notes = lireTab(nbe);

    Terminal.ecrireStringln("Notes obtenues: ");

    afficheTab(notes);

    Terminal.ecrireString("Moyenne:  " );

    Terminal.ecrireDoubleln(moyenneTab(notes));

}
```

Moyenne des notes dans un tableau(4)

Problème : lire un tableau de notes, calculer et afficher la moyenne.

```
public static void main(....) {

    int  nbe = lirePos();

    double [] notes = lireTab(nbe);

    Terminal.ecrireStringln("Notes obtenues: ");

    afficheTab(notes);

    Terminal.ecrireString("Moyenne:  " );

    Terminal.ecrireDoubleln(moyenneTab(notes));

}
```



Programmez avec des sous-programmes !

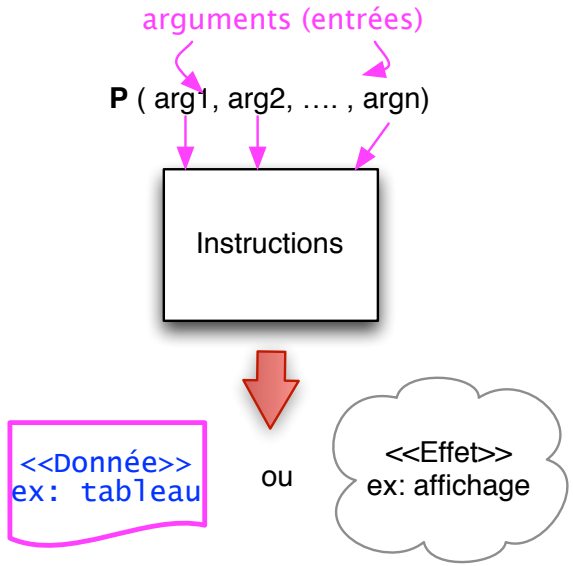
Les sous-programmes servent à structurer les programmes.

- A chaque fois que cela est utile : utilisez des sous-programmes :
  1. Identifier une sous-tâche ;
  2. Identifier ses entrées/sorties + écrire son code ;
  3. invoquer partout où cette sous-tâche est nécessaire ;
  4. ré-utiliser le sous-programme dans d'autres programmes...
- Sous-programmes prédéfinis : dans les « bibliothèques » d'un langage.
  - à utiliser autant que possible !

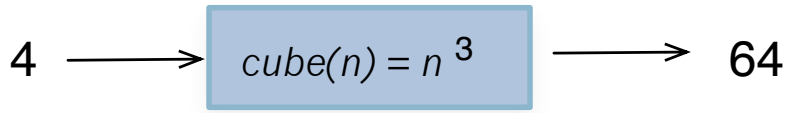
2. Déclarer et utiliser ses sous-programmes

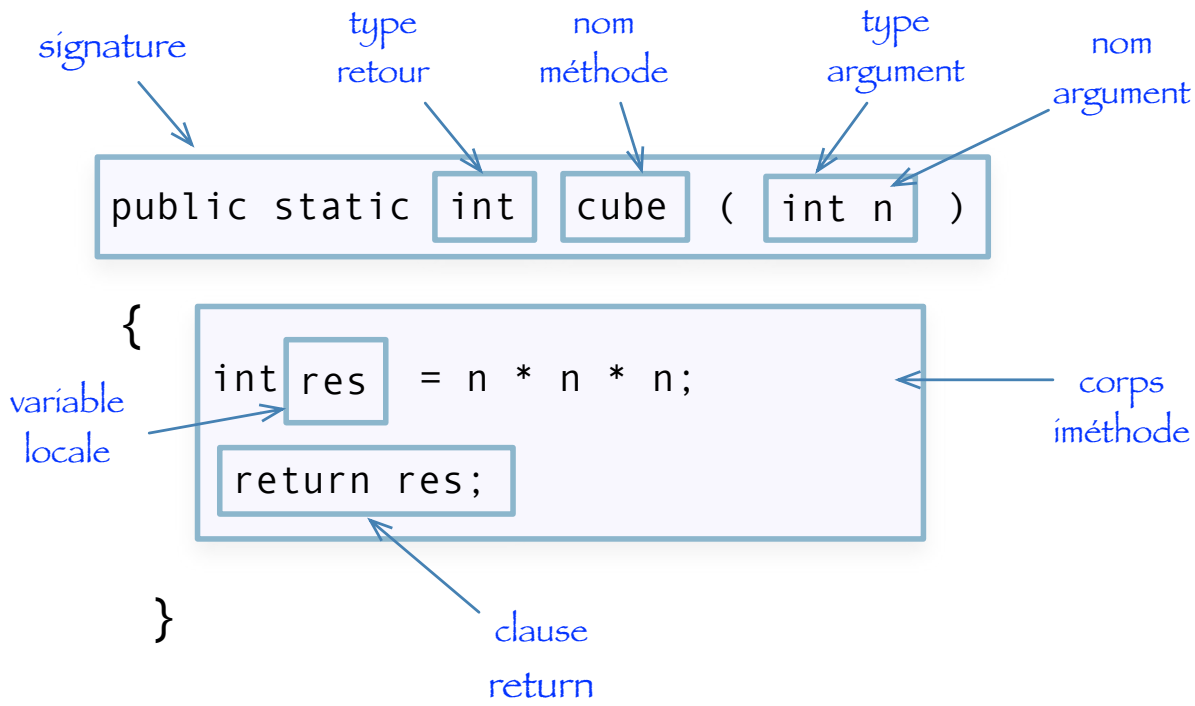
Sous-programme : entrées + exécution + sorties/retour

1. prend un ou plusieurs paramètres ;
2. si invoqué : exécute une suite d'instructions ;
3. produit un résultat ou un comportément.



Anatomie d'un sous-programme





Sous-programmes en Java

- Appelés méthodes.
- Deux sortes :
  - Statiques : (mot-clé `static`) $\Rightarrow$ non « attachées » à un objet.
    Nous utiliserons uniquement celles-ci !
  - Non statiques : ou « méthodes d'objets ». A voir plus tard (NFA032).
- beaucoup de méthodes prédéfinies :
  - `Math.random()`, `Math.abs()`, `Math.sqrt()`,
  - `System.out.print()`, `Integer.parseInt()`, etc.

Fonctions et procédures

Dépend du type de retour :

- Fonction :
  - produit une valeur (à récupérer par l'appelant),
  - type de retour $\neq$ `void`
  - Exemple : `static int Math.min(int x, int y)`

    $\Rightarrow$ produit un int
- Procédure :
  - produit un comportément (sans donnée à recupérer).
  - type de retour = `void`
  - Exemple : `static void Terminal.ecrireInt(int x)`

    $\Rightarrow$ affiche un entier

Vocabulaire

- Arguments, paramètres : entrées d'un sous-programme ;
- Appel, invocation : exécution instructions du sous-programme sur des arguments donnés ;
- selon type de retour :
  - Fonction :
    - ★ type de retour $\neq$ `void`
    - ★ produit une valeur (à récupérer par l'appelant),
  - Procédure :
    - ★ produit un comportément (sans donnée obtenue).
    - ★ type de retour = `void`

Déroulement d'un appel : M invoque P

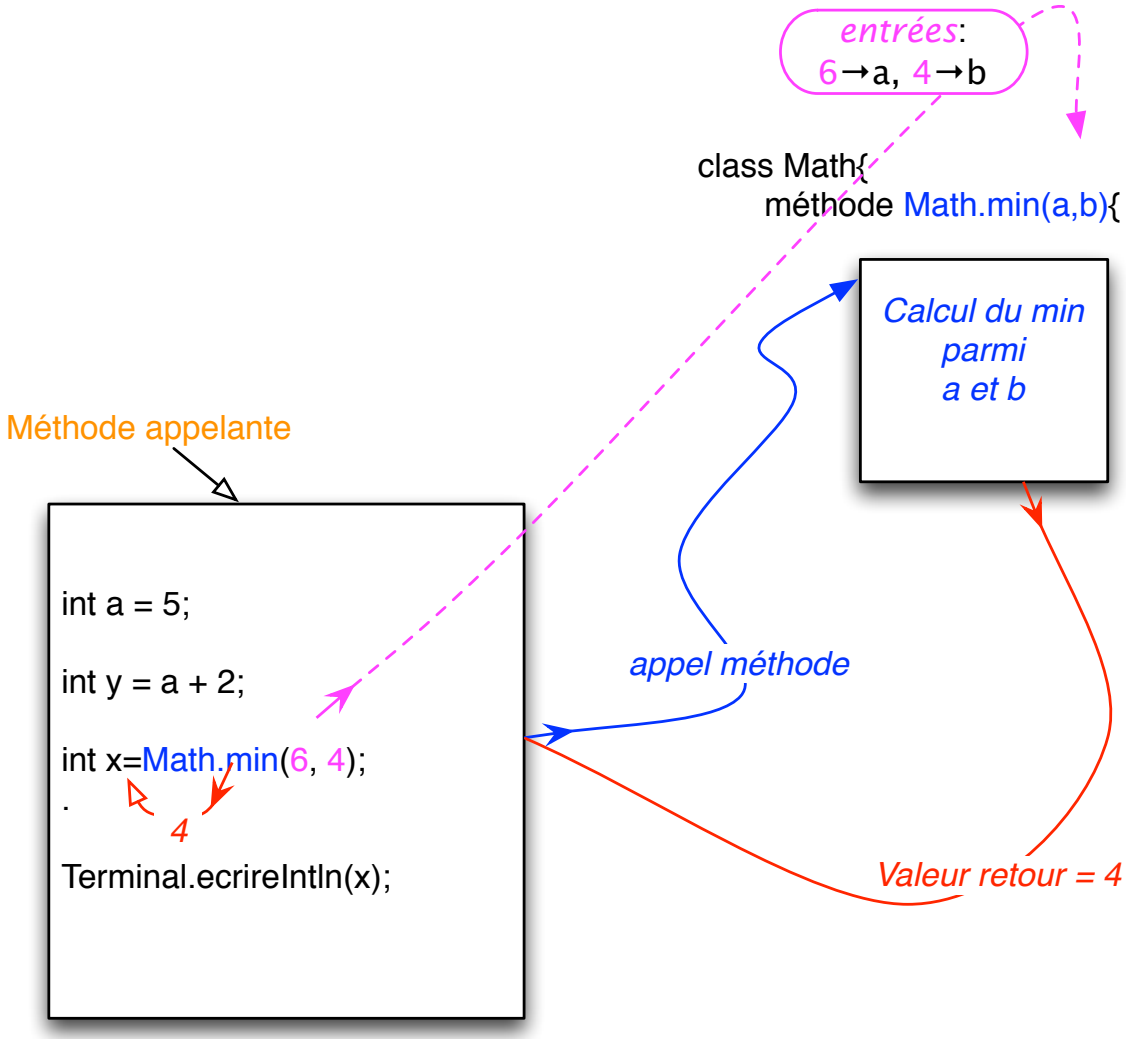
Vocabulaire :

- méthode appelante : celle qu'invoque le sous-programme P ;

Déroulement appel :

1. Interruption exécution méthode appelante M ;
2. Contrôle $\overset{\text{passé}}{\Longrightarrow}$ à P ;
3. Arguments de P $\overset{\text{affectés}}{\Longleftarrow}$ par *valeurs* d'invocation dans M ;
4. Exécution code de P ;
5. Retour $\overset{\text{(résultat)}}{\Longrightarrow}$ vers M ;
6. Reprise contrôle + exécution M (avec résultat obtenu).

Méthode appelle sous-programme Math.min



Le rôle des paramètres

Paramètres

Ce sont les entrées du sous-programme : des valeurs passées en paramètre lors de chaque appel.

- nécessaires pour l'exécution du sous-programme.
- peuvent être différents à appel ;

```
Terminal.ecrireInt(5);
Terminal.ecrireInt(10);
```

- Certains sous-programme ne prennent pas d'arguments : `Terminal.lireDouble()`.

Où déclarer + utiliser des sous-programmes

Déclarer : nom, paramètres, corps ;

Utiliser : autant d'appels que nécessaire, à partir d'autres méthodes, et avec leurs arguments.

```
public class <nom-du-programme> {

   <déclaration-des-sousprogrammes>

   public static void main (String[] args) {

      <déclarations-instructions+appels>
   }
}
```

Exemple de fonction

```
public class ValAbsFunc {
   // Declaration
   static int valeurAbsolue(int n){ // debut
      if (n >= 0) { return n;
      } else { return -n;}
   }                                        // fin

   public static void main (String args[]) {
      int  x = -7;
      Terminal.ecrireString("Valeur absolue de "+x+": ");
      // Appel
      Terminal.ecrireIntln(valeurAbsolue(x));
   }
}
```

Clause return et fonctions

Fonctions : doivent toujours finir leur exécution par un return

```
static int valeurAbsolue(int n){
    int res;
    if (n > 0) { res = n;
    } else if(n < 0){ return -n;
    } else { return 0;}
}
```

Si `n > 0`, pas d'instruction `return`.
⇒ erreur (compilation) :

```
> javac ValAbsFunc3.java
ValAbsFunc3.java:11: missing return statement
```

Exemple de procédure

```
public static void afficheTriangleDecroissant(){
  for(int i=1; i<6; i++){
    for (int j=5; j>=i; j--){
        System.out.print(j + "␣");
    }
    System.out.println();
  }
}
```

- cette méthode ne produit pas de donnée ;
- produit un comportément (dessiner, afficher) ;
- type de retour *Rightarrow* `void` ;
- Autre particularités : pas d'arguments + invoque d'autres méthodes.

Appels imbriqués

Une méthode peut invoquer d'autres méthodes :

```
static boolean estMajuscule(char c){
    return ('A' <= c && c <= 'Z');
}
static boolean estMinuscule(char c){
    return ('a' <= c && c <= 'z');
}
static boolean estLettre(char c){
    return ( estMajuscule(c) || estMinuscule(c) );
}
public static void main(String[] args) {
    char c1 = 'a';
    if (estLettre(c1))
        System.out.println(c1+"␣est␣une␣lettre");
    else
        System.out.println(c1+"␣n'est␣est␣pas␣une␣lettre");
}
```

Visibilité des variables d'un sous-programme

Variables d'un sous-programme :

- ses paramètres,
- ses variables déclarées localement,

Visibles uniquement dans le sous-programme :

- un sous-programme ne peut utiliser que ses variables locales + paramètres ;
- toute autre variable est considérée inconnue (erreur compilation)

Variables méthode m $\approx$ visibles uniquement par m

Variables d'un sous-programme : exemple (1)

Combien de variables dans ce sous-programme ?

```
static int valeurAbsolue(int n){
    int res;
    if (n > 0) { res = n;
    } else { res = -n;
    }
    return res;
}
```

2 variables : paramètre `n` et `res`.

Variables d'un sous-programme : exemple (2)

```
static int plusUn(int x){
  int r = x+1;
  return r;
}
public static void main (String [] args){
  int x = 3;
  int y = plusUn(x*2);
  Terminal.ecrireStringln("Resultat: "+y);
  Terminal.ecrireStringln("valeur finale x: "+x);
```

- Quelles variables par méthode ?
- `x` est-elle commune aux deux méthodes ?
- Affichages ?

Variables d'un sous-programme : exemple (3)

- Variables :
  - méthode `plusUn` ⇒`x` et `r` ;
  - méthode `main`⇒ `x` , `y` et `args` ;
- Chaque x est locale à "sa" méthode ⇒2 variables x différentes.
- Affichages :

```
Resultat : 7
valeur finale x : 3
```

3. Exemples

Exemples : fonctions et procédures

Ré-visitons le problème de départ : lecture d'un tableau de notes, affichage notes, calcule et affichage de la moyenne.

1. Lire un nombre de notes strictement positif (boucle).
2. Créer un tableau de cette taille et l'initialiser avec autant de notes lues.
3. Afficher l'ensemble de notes.
4. Calculer et afficher la moyenne.

Exemple : fonction de lecture d'un entier positif

Problème : lire un entier strictement positif
Solution : fonction avec boucle de lecture + validation
retourne : l'entier (valide)

```
static int lirePos(String message) {
  int n = 0;
  while (n<=0)  {
    Terminal.ecrireString(message);
    n = Terminal.lireInt();
    if (n<=0) {
      Terminal.ecrireStringln("Le nombre doit etre positif"
    }
   }
   return n;
}
```

A quoi correspond le paramètre `message` ?

Fonction de lecture d'un entier positif : appel

```
/* Lit un entier strictement positif en affichant
 * message pour inviter  a la saisie  */
static int lirePos(String message) {
  int n = 0;
  while (n<=0)  {
    Terminal.ecrireString(message);
    n = Terminal.lireInt();
    if (n<=0) {
      Terminal.ecrireStringln("Le nombre doit etre positif"
    }
   }
   return n;
}
public static void main (String args[]) {
  int nbeNotes = lirePos("Nombre de notes? ");
  ....
```

Fonction de calcul moyenne d'un tableau

Problème : calculer la moyenne d'un tableau de notes
Solution : fonction de parcours+calcul
param : tableau de notes
retourne : moyenne calculée

```
  /* Calcule la moyenne de composantes dans t
   */
static double moyenneTab (double [] t) {
    double somme = 0;
    for (int i=0; i< t.length; i++) {
          somme = somme + t[i];
    }
    return somme/t.length;
}
```

Fonctions ⇒ ni saisie ni affichage

Principe de programmation : sauf fonctions de lecture, une fonction :

- ne fait pas de lecture ⇒ ses entrées = paramètres ;
- ne fait pas d'affichage ⇒ ses sorties = valeur de retour (return) ;

```
static double moyenneTab (double [] t) {
    double somme = 0;
    for (int i=0; i< t.length; i++) {
          somme = somme + t[i];
    }
    return somme/t.length;
}
```

Ne lit pas le tableau ⇒ le prend en paramètre ;
N'affiche pas la moyenne ⇒ la retourne en résultat !

Procédure affichage d'un tableau

Problème : afficher un tableau de notes
Solution : boucle parcours+affichage (pas de calcul)
param : tableau de notes
pas de valeur retour ⇒ procédure

```
  /* Affiche composantes d'un tableau de notes
   */
static void afficheTabNotes (double [] t) {
    for (int i=0; i<t.length; i++)  {
      Terminal.ecrireStringln("note␣"+ (i+1)+":␣"+ t[i]);
    }
}
```

Remarque : une procédure/fonction ne lit pas ses entrées ⇒ les prend en paramètre ;

Fonction création + lecture tableau

Problème : créer un tableu de taille n et l'intialiser par lecture
Solution : création puis parcours+lecture
param : n (taille tableau)
valeur retour : tableau crée et intialisé

```
/* Creation et lecture d'un tableau taille n
 * Retourne: tableau avec composantes lu
 */
static double [] lireTab (int n) {
    double [] t = new double[n];
    for (int i=0; i<= t.length -1; i++)  {
         Terminal.ecrireString("Une note? ");
         t[i] = Terminal.lireDouble();
    }
    return t;
}
```

Méthode main

```
public class moyenneNotesMeth {
   /*  Declaration des sous-programmes
       .....
   */
   public static void main (String args[]) {
     int nbeNotes = lirePos("Nombre de notes? ");
     double [] notes =  lireTab(nbeNotes);
     Terminal.ecrireStringln("Notes obtenues: ");
     afficheTabNotes(notes);
     Terminal.ecrireString("La moyenne des notes est: ");
     Terminal.ecrireDoubleln(moyenneTab(notes));
  }
}
```

Programme complet

```
public class moyenneNotesMeth {
   static double [] lireTab (int nombreEl) {
      double [] t = new double[nombreEl];
      for (int i=0; i<= t.length -1; i++)  {
                Terminal.ecrireString("Une note? ");
                t[i] = Terminal.lireDouble();
      } return t;
   }
   static int lirePos(String message) {
      int n = 0;
      while (n<=0)  {
          Terminal.ecrireString(message);
          n = Terminal.lireInt();
          if (n<=0) {
            Terminal.ecrireStringln("Le nombre doit etre positif");
          }
      }   return n;
   }
   static void afficheTabNotes (double [] t) {
      for (int i=0; i<t.length; i++)  {
       Terminal.ecrireStringln("note "+ (i+1)+": "+ t[i]);
      }
   }
   static double moyenneTab (double [] t) {
      double somme = 0;
      for (int i=0; i< t.length; i++) {
           somme = somme + t[i];
      } return somme/t.length;
   }
   public static void main (String args[]) {
       int nbeNotes = lirePos("Nombre de notes? ");
       double [] notes =  lireTab(nbeNotes);
```

Lire des notes comprises entre 0 et 20

Problème : valider les notes lues (comprises entre 0 et 20)
Solution : boucle de lecture+ validation d'un nombre
params : message affiché + bornes inférieur, supérieur ;
valeur retour : nombre compris entre inf et sup

```
static double lireNote(String me,double inf,double sup) {
    while (true)  {
        Terminal.ecrireString(me);
        double n = Terminal.lireDouble();
        if (n<inf || n> sup) {
          Terminal.ecrireStringln("Entree invalide");
        } else return n;
    }
}
```

Que faut-il modifier pour utiliser cette méthode ?

main appelle lireTab qui appelle lireNote . . .

Dans la fonction de lecture du tableau de notes :

- `Terminal.lireDouble()` devient
  ⇒ `lireNote("Une note? ", 0, 20)`

```
static double [] lireTab (int n) {
    double [] t = new double[n];
    for (int i=0; i<= t.length -1; i++)  {
      //Terminal.ecrireString("Une note? ");
      t[i] = lireNote("Une note? ", 0, 20);
    }
   return t;
}
```

Remarquez le commentaire : plus besoin d'afficher explicitement le message "Une note ? ".

Erreurs fréquentes (1)

Un sous-programme qui reçoit un paramètre, ne doit pas mofifier ce paramètre avant de l'utiliser (contradiction car paramètre = entrée).

```
  static int valeurAbsolue(int n){
     n = 7;    // Non!!
     if (n > 0) { return n;
     } else if (n < 0) { return -n;
     } else { return 0;
     }
  }

  public static void main (String args[]) {
     int x = -3;
     Terminal.ecrireString("Valeur␣absolue␣de␣"+x +":");
     Terminal.ecrireIntln(valeurAbsolue(x));
  }
}
```

Erreurs fréquentes (2)

Un sous-programme qui reçoit un paramètre, ne doit pas saisir sa valeur (contradiction, car le paramètre est déjà l'entrée).

```
  static int valeurAbsolue(int n){
     Terminal.ecrireString("Donnez un entier ");
     n = Terminal.lireInt(); // Non!!
     if (n > 0) { return n;
     } else if (n < 0) { return -n;
     } else { return 0;
     }
  }

  public static void main (String args[]) {
     int x = -3;
     Terminal.ecrireString("Valeur absolue de "+x +":");
     Terminal.ecrireIntln(valeurAbsolue(x));
  }
}
```

Erreurs fréquentes (3)

Une fonction retourne un résultat et ne doit pas s'occuper de son affichage.

```
static int valeurAbsolue(int n){
     if (n > 0) { Terminal.ecrireInt(n);// Non!!
    } else if (n < 0) { return -n;
                        Terminal.ecrireInt(-n);// Non!!
    } else {Terminal.ecrireInt(0); return 0;// Non!!
    }
}
public static void main (String args[]) {
    int x = -3;
    Terminal.ecrireString("Valeur absolue de "+x +":");
    Terminal.ecrireIntln(valeurAbsolue(x));
}
```

Ce programme ne se comporte pas correctement. Pourquoi ?

Erreurs fréquentes (4)

Penser qu'il y a un lien entre le nom donné au paramètre et celui de la variables de l'appel (ici x).

```
  static int valeurAbsolue(int x){
      if (x > 0) { return x;
    } else if (x < 0) { return -x;
    } else {  return 0;
    }
  }
  public static void main (String args[]) {
    int x = -3;
    Terminal.ecrireString("Valeur absolue de "+x +":");
    Terminal.ecrireIntln(valeurAbsolue(x));
  }
}
```

le paramètre x du sous-programme, et la variable x du main, sont de variables locales distintes et indépendantes.

Erreurs fréquentes (resumé)

- Un sous-programme ne saisit pas les valeurs de ses paramètres.
- Un sous-programme ne modifie pas les valeurs de ses paramètres (avant de s'en servir).
- Une fonction n'affiche pas son résultat. Elle le renvoie.
- Une fonction ne modifie jamais ses paramètres.
- Les paramètres formels (ceux déclarés dans la méthode) sont des variables locales et donc, sont indépendantes des variables utilisées pour un appel.

Procédure ou Fonction ?

Si l'on veut écrire un sous-programme, comment savoir s'il doit correspondre à une procédure ou à une fonction ?
Comment savoir quels sont les arguments ?

- Le sous-programme doit-il calculer un résultat ? Quel est son type ? (fonction + type de retour)
- Doit-il faire des affichages sans calculs ? (procédure)
- Quelles sont les données nécessaires à son exécution ? Quels sont leurs types ? (paramètres)
- Si un sous-programme doit réaliser des entrées/sorties et des calculs, il vaut mieux le couper en autant de morceaux que nécessaire afin de séparer clairement les fonctions des procédures.

5. 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);
}
```

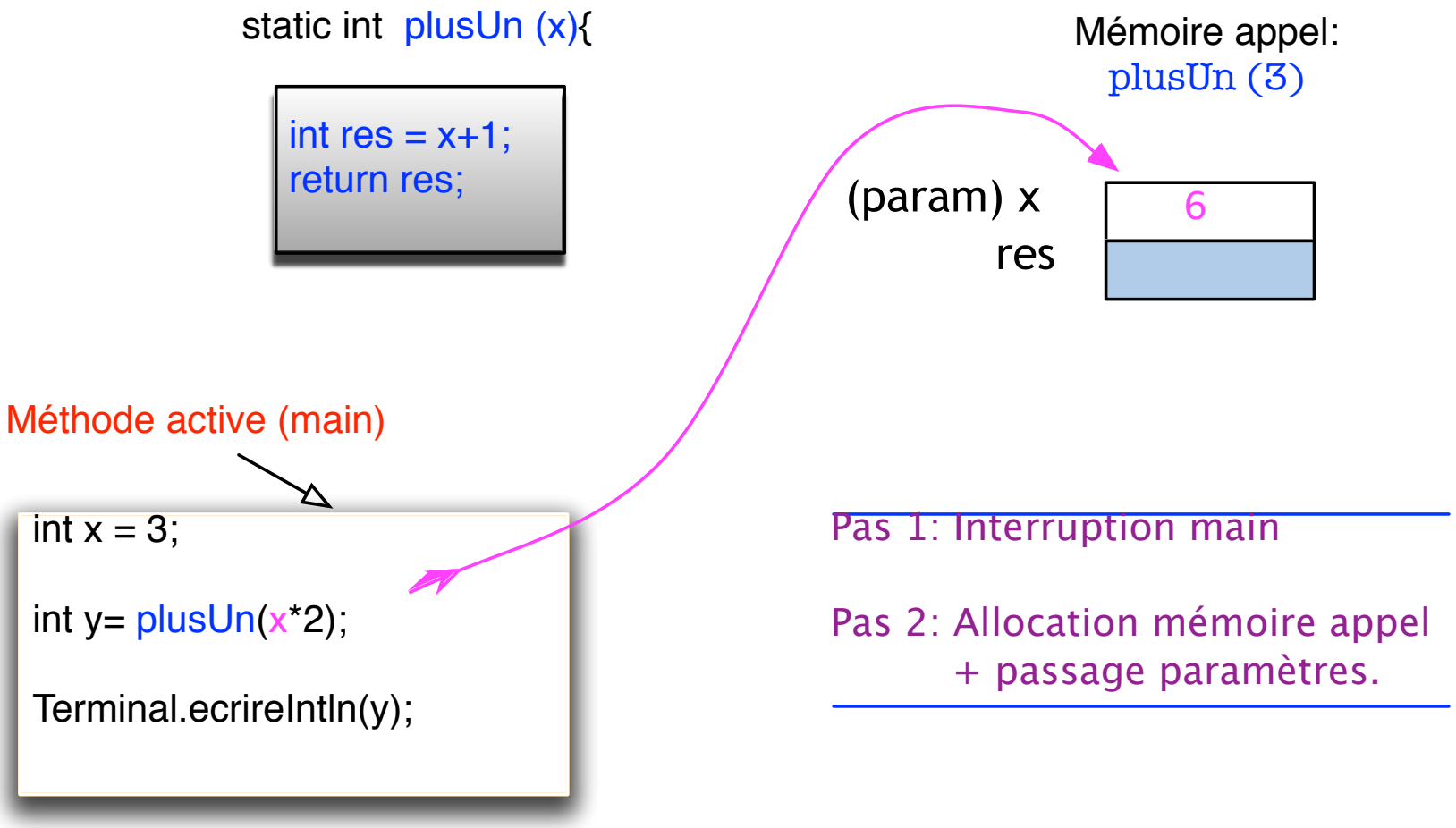
Avec quels arguments se fait l'appel ? ⇒ plusUn(6)

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)

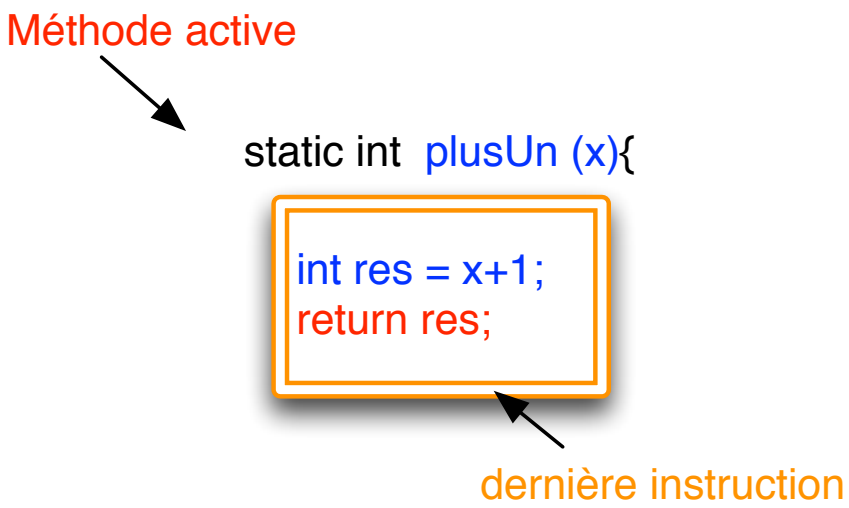
Exécution appel plusUn



Pas 1: Interruption main

Pas 2: Allocation mémoire appel
+ passage paramètres.

Exécution appel plusUn (2)



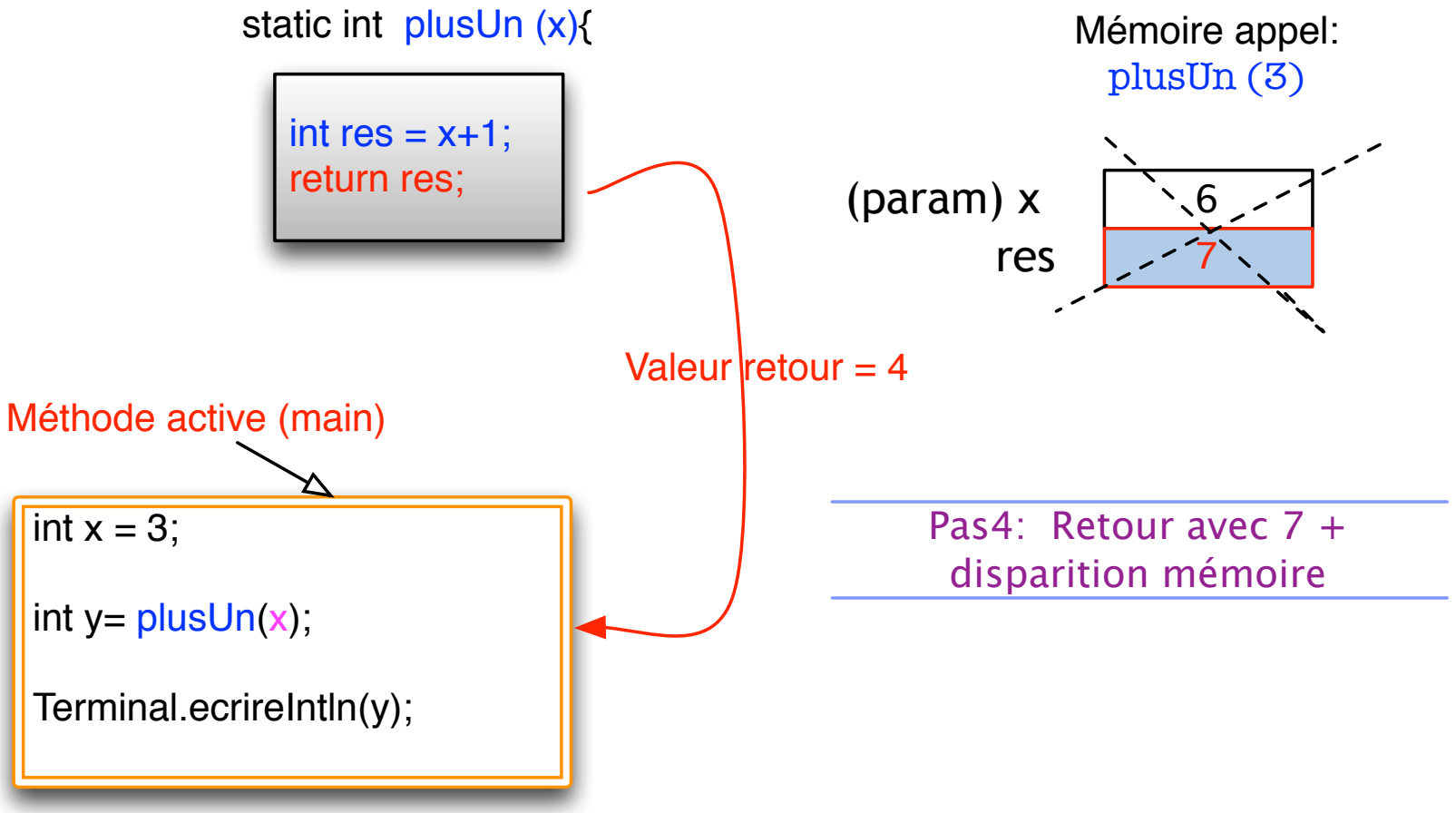
Mémoire appel:
plusUn (3)

(param) x	6
res	7

```
int x = 3;

int y= plusUn(x * 2);

Terminal.ecrireIntln(y);
```

Pas 3: Exécution plusUn

Exécution appel plusUn (3)



Passage de paramètres ⇒ recopie de valeurs

```
static int P(int n){ ...}
static void main (...){
  P(3); //  appel --> copie 3 vers n dans memoire de P
}
```

Passage de paramètres

Copie des valeurs de l'appelant ⇒ mémoire locale appelé

- si appel `P(x)` : copie de 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.

Que peut faire m de plus ou de moins selon le cas ?

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` ?

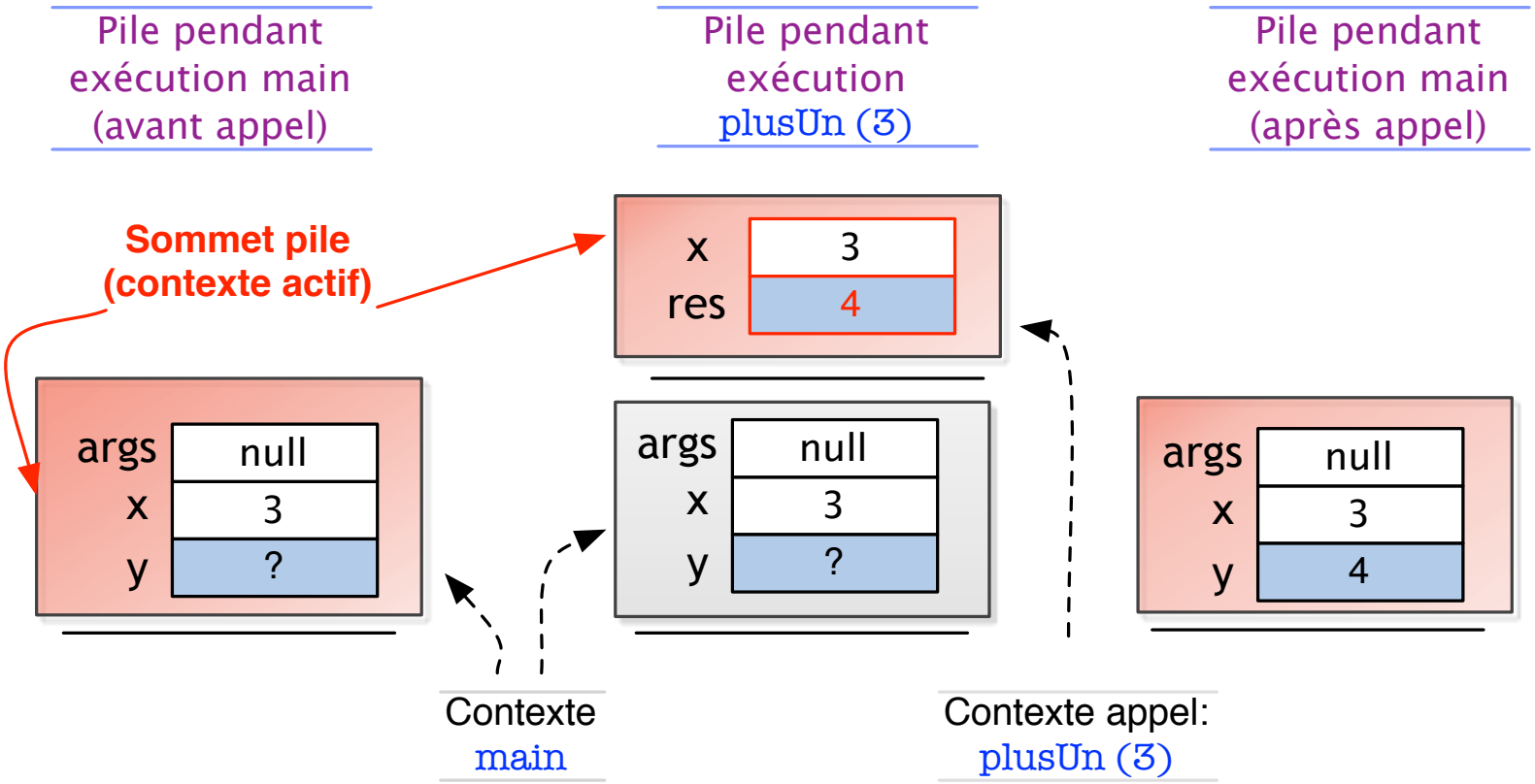
Pile d'exécution

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.



6. Paramètres de type référence

Procédure `inversionTab`

Considérons maintenant une version procédurale de cette méthode :

```
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;
   }
}
```

- est-ce une procédure ou une fonction ? pourquoi ?
- différences dans le code des deux méthodes ?

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 ?

Procédure avec argument de type référence

Que se passe-t-il si :

- une procédure modifie explicitement son argument ?
- cela change la valeur de la variable passée par la méthode appelante ? (c.a.d., la mémoire de la méthode appelante ?)
- différence selon que l'argument est de type primitif ou référence ?

Retour sur le passage de paramètres

Le passage de paramètres en Java se fait « par valeur » :

⇒ lors d'un appel `m(x)`, on passe à m :
la valeur contenue dans la variable x.

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

Que peut faire m de plus ou de moins selon le cas ?

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 ?

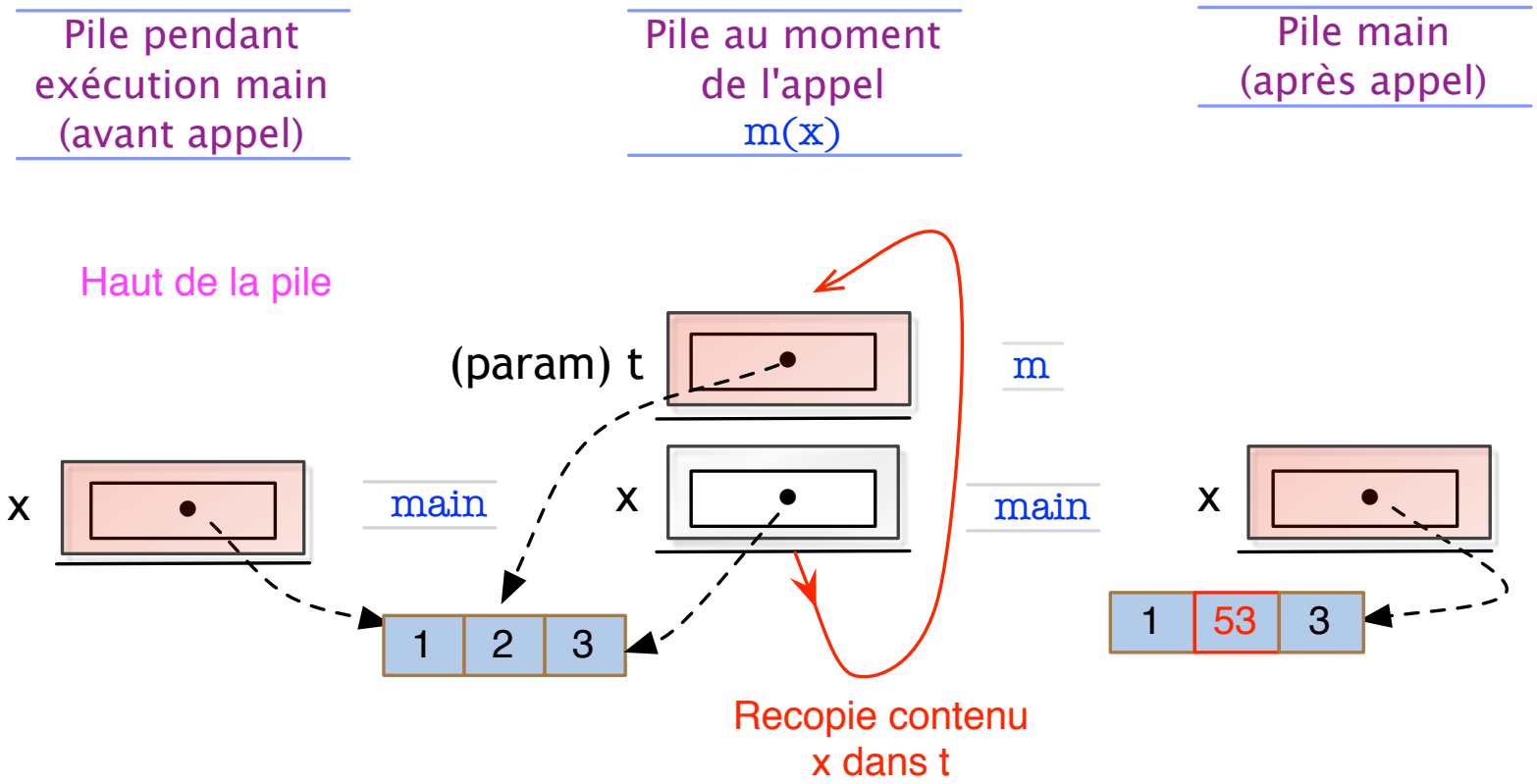
Passage avec type référence (2)

```
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

7. Faire échouer un sous-programme

Fonction partielles

Parfois, une fonction n'a pas de résultat défini pour certains cas.

Exemple : la fonction qui renvoie l'indice où se trouve un élément dans un tableau, n'est pas définie si l'élément n'est pas trouvé.

```
static int indiceDe(int n, int [] t) {
   for (int i=0; i<t.length; i++){
      if (t[i]==n) return i;
   }return ??
}
```

On peut dans ce cas déclencher une erreur. Le sous-programme est arrêté avec une erreur irrecupérable.

Les exceptions

- Ce sont des erreurs que l'on peut déclarer avec un nom,
- qu'on peut déclencher, mais aussi gérer.
- Elles provoquent l'arrêt de l'exécution de l'instruction qui les a déclenchées, (`throw new nom-exception()`)
- et si elles ne sont pas traités, celui de tout le programme.

Les exceptions

```
class Chercher{
    static int indiceDe(int n, int [] t) {
      for (int i=0; i<t.length; i++){
         if (t[i]==n) return i;
      }
      throw new NonTrouve();  // Déclenchement
    }

    public static void main(String[] argv){
        int x; int [] tab = {1,3,9};
        x = indiceDe(2,tab);
        Terminal.ecrireString("2 est a l'indice "+ x);
    }
}
class NonTrouve extends Error{} // Déclaration
```

Les exceptions

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

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

```
> java Chercher
java.lang.ErrorException: / NonTrouve
        at Chercher.main(Chercher.java:4)
```

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

8. 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 [])`.