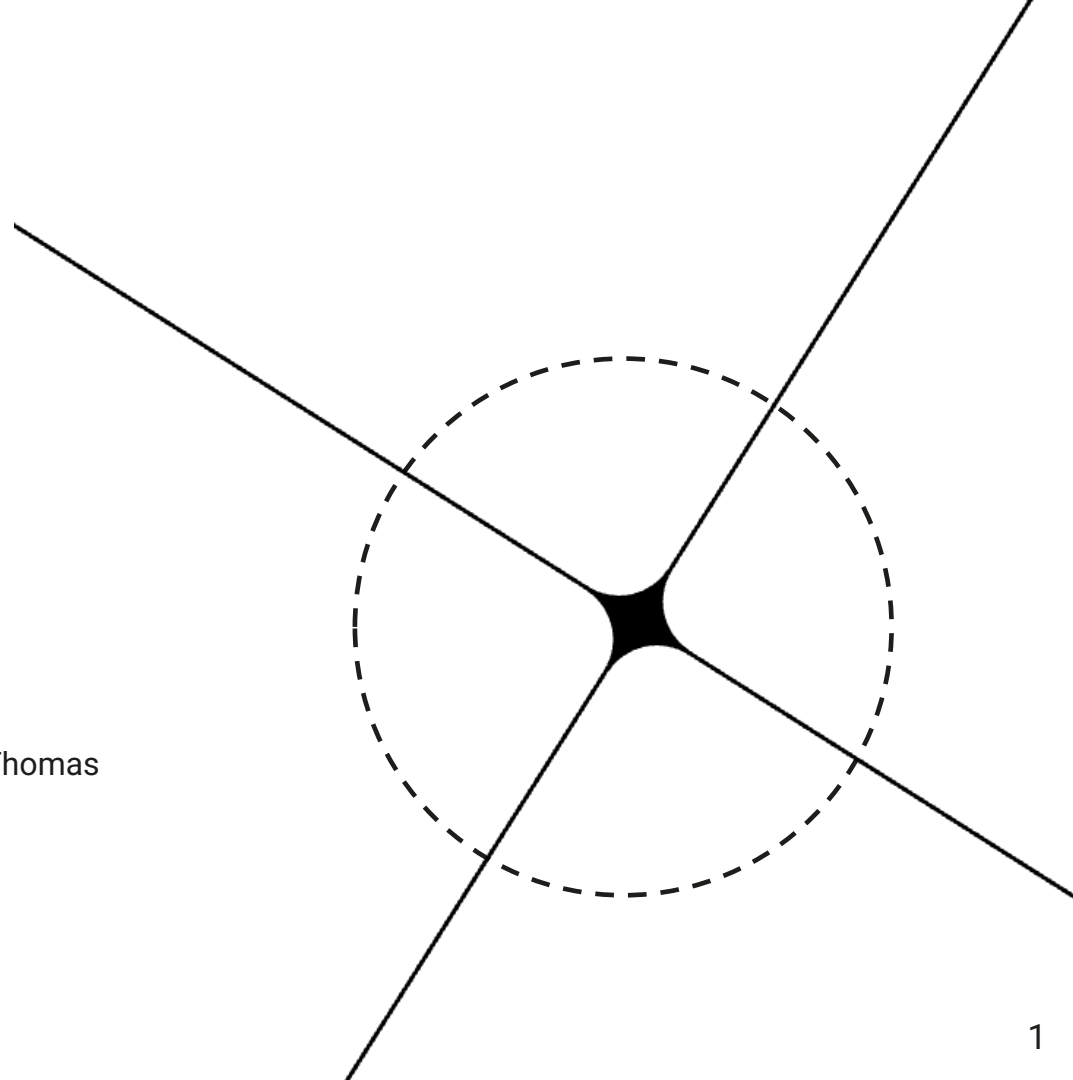
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Cadre d'appel, passage d'arguments en Java

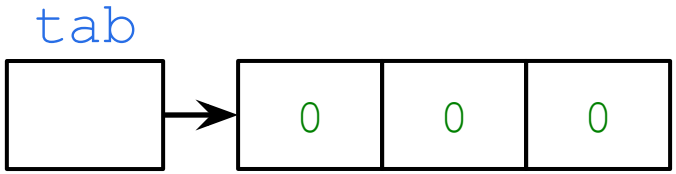
Alexandre Nolin, Julien Romero, Pierre Sutra, Gaël Thomas



Rappel CM1 : Type référence versus type primitif

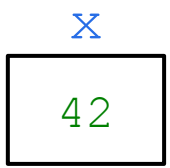
- Variable de type référence : contient une référence vers une structure de données
  - Tableaux et String
  - Très commun en Java !

```
int[] tab = new int[3];
```

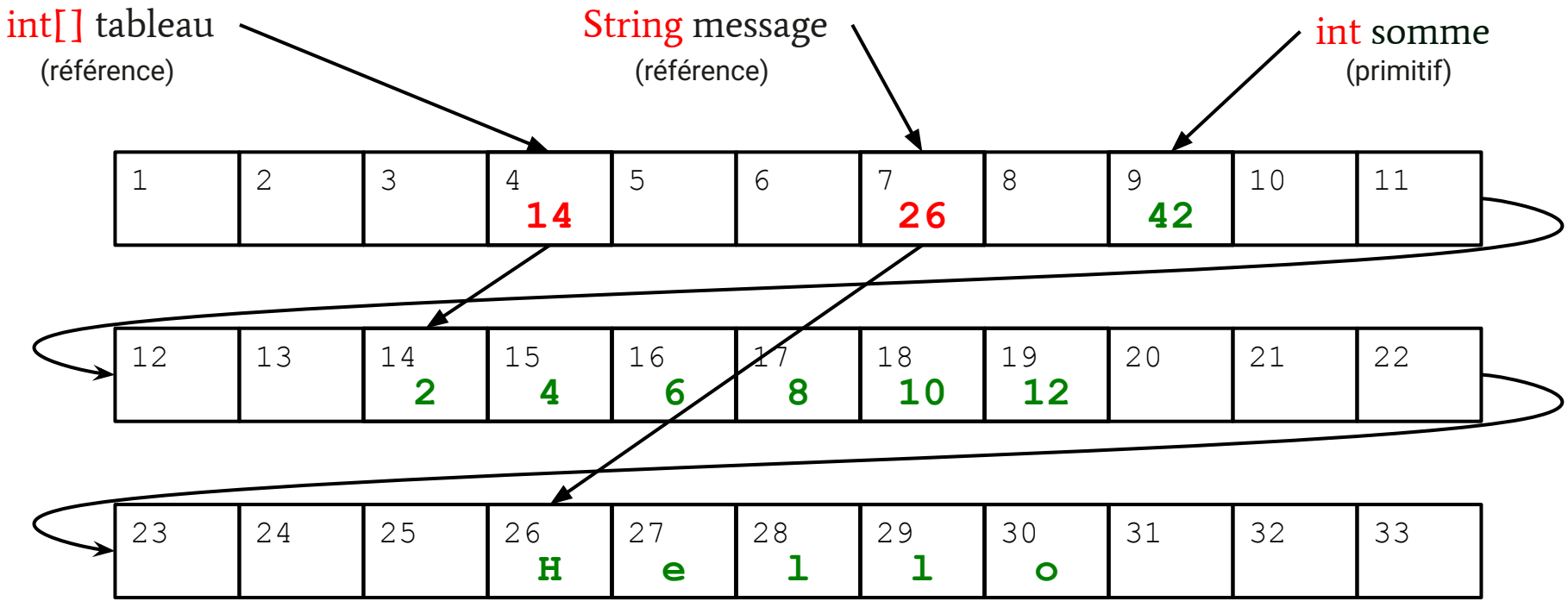


- Variable de type primitif : contient une valeur
  - boolean, byte, short, char, int, long, float et double
  - On n'a pas de référence pour les types primitifs !

```
int x = 42;
```



Rappel : Type référence versus type primitif dans la mémoire



Une variable de type référence contient une adresse mémoire vers le contenu en lui-même, une variable de type primitif contient la valeur elle-même.

Variables locales (1/2)

- Variable locale = variable définie dans une méthode
  - **N'existe que le temps de l'invocation** de la méthode
  - Il en va de même des paramètres de la méthode

- Lors d'une invocation de méthode, l'environnement d'exécution:
  - Crée un **cadre d'appel** pour accueillir les variables locales et les paramètres
  - Crée les variables locales/paramètres dans le cadre
  - Affecte **par valeur** les paramètres

- À la fin de l'invocation, l'environnement d'exécution
  - Détruit le cadre, ce qui détruit les variables locales/paramètres

Variables locales (1/2)

- Variable locale = variable définie dans une méthode
  - **N'existe que le temps de l'invocation** de la méthode
  - Il en va de même des paramètres de la méthode
- Lors d'une
  - Crée
  - Crée
  - Affe
- À la fin de l'invocation, l'environnement d'exécution
  - Détruit le cadre, ce qui détruit les variables locales/paramètres

cadre d'appel
=
call frame en anglais

Variables locales (2/2)

- Avant l'entrée dans `main`
  - Le tableau des arguments est initialisé

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

Tableau des arguments

Variables locales (2/2)

- À l'entrée dans `main`
  - Création cadre de main

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

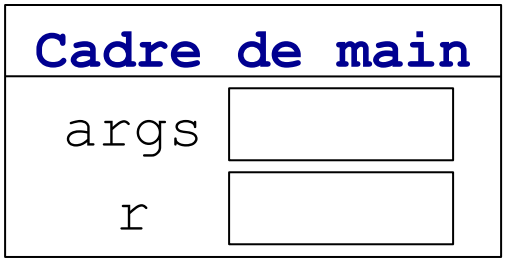
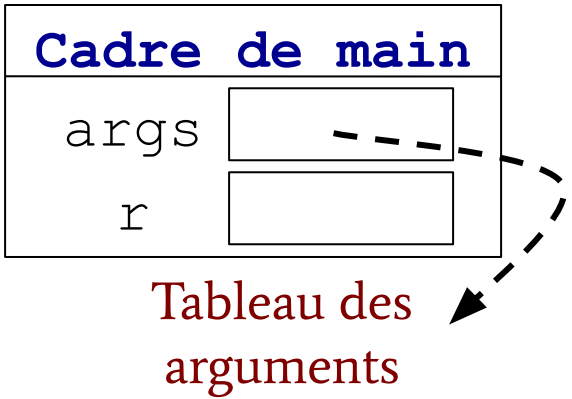


Tableau des arguments

Variables locales (2/2)

- À l'entrée dans `main`
  - Création cadre de main
  - Affectation paramètre `args` (référence vers tab. arguments)

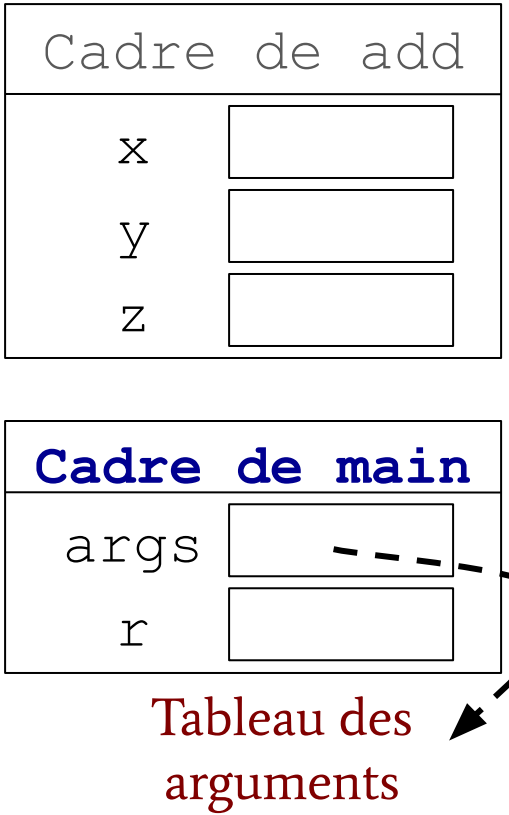
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À l'entrée dans `add`
  - Création cadre `add`

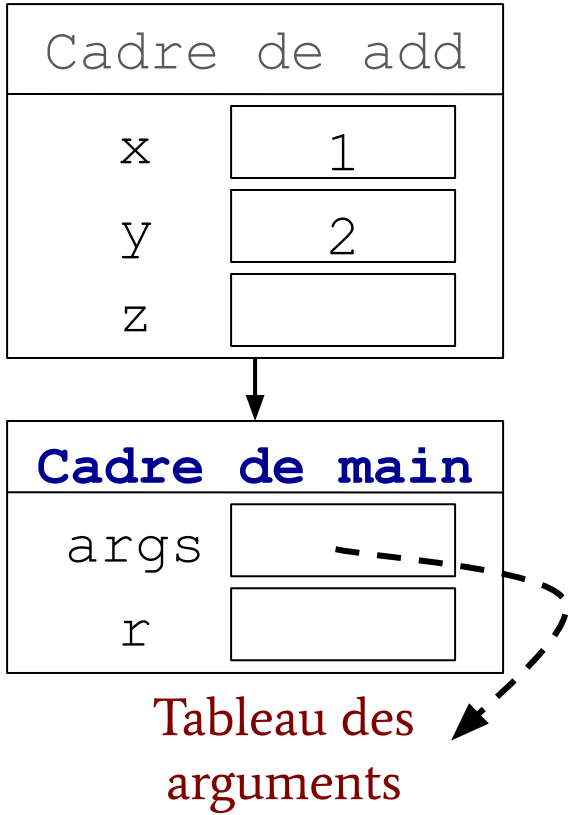
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À l'entrée dans `add`
  - Création cadre `add`
  - Affectation paramètres

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

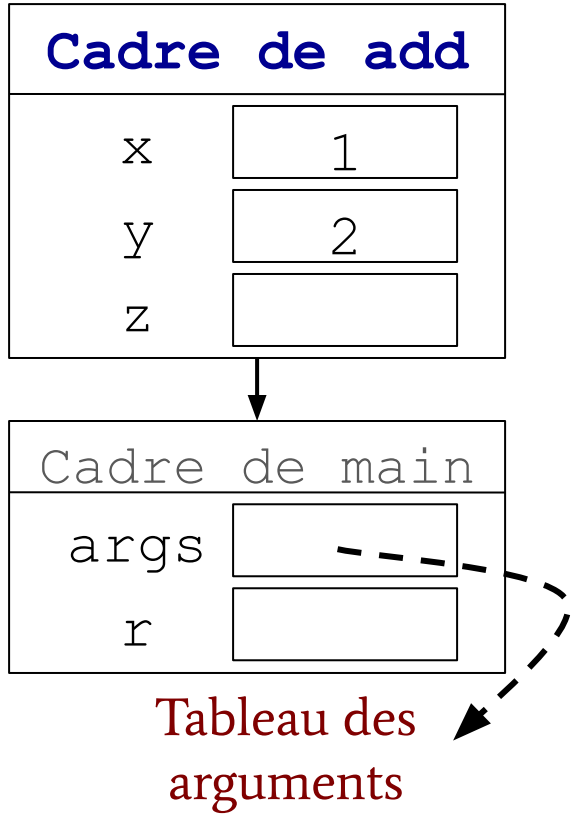


Variables locales (2/2)

- À l'entrée dans `add`
  - Activation du cadre de `add`

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

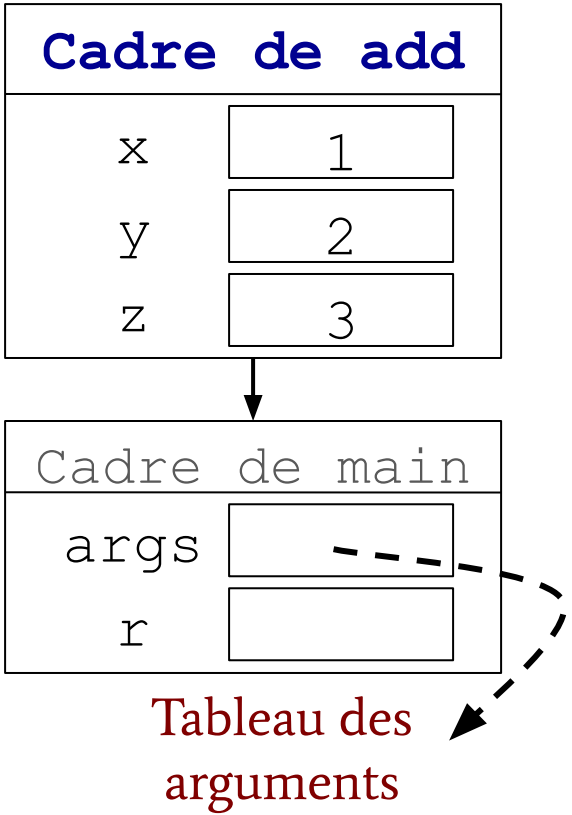
Cadre actif
en bleu/gras



Variables locales (2/2)

- Pendant `add`
  - Utilisation des variables locales du **cadre actif**

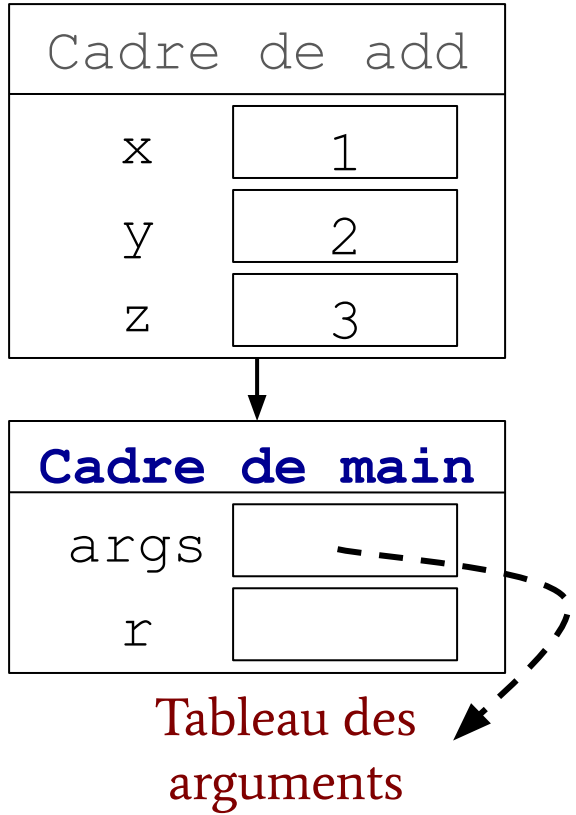
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À la sortie de `add`
  - Activation cadre précédent

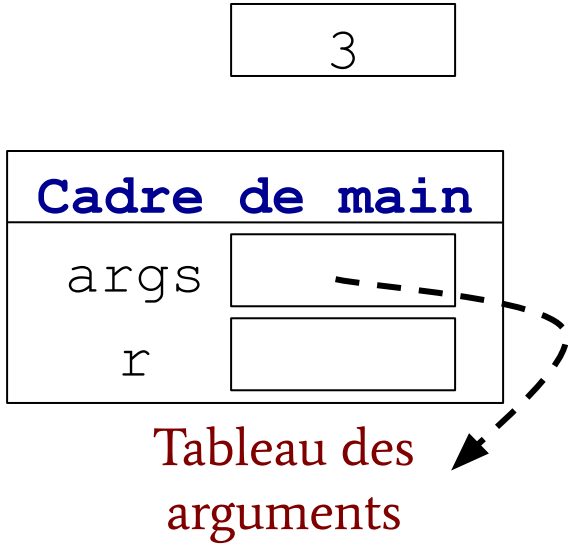
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

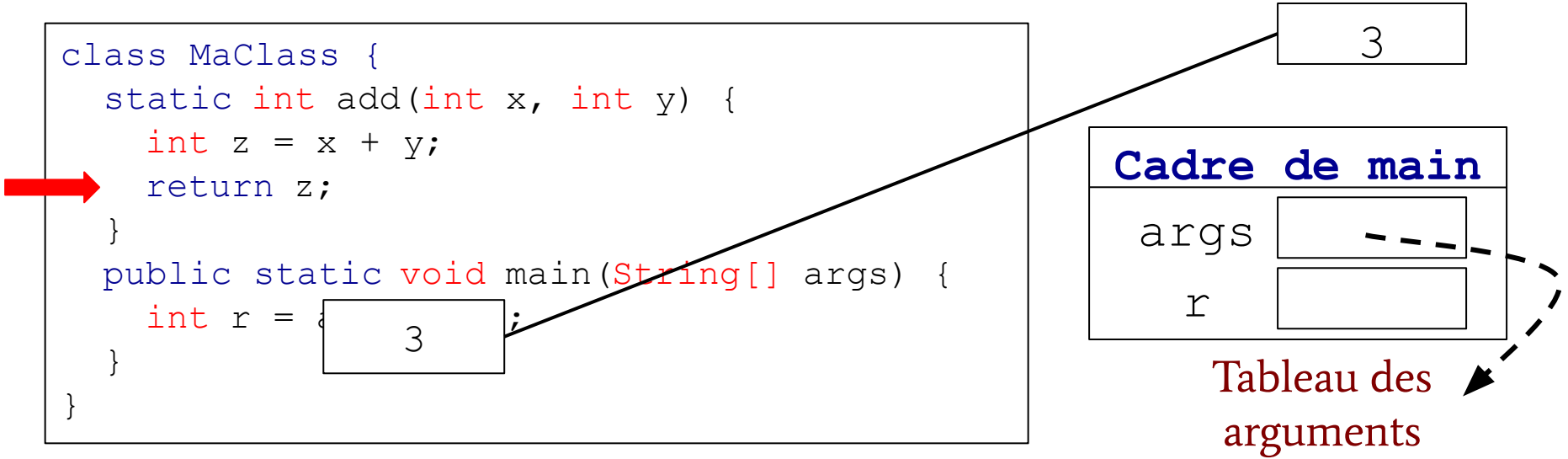
- À la sortie de `add`
  - Activation cadre précédent
  - Suppression cadre précédent + récupère valeur de retour

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

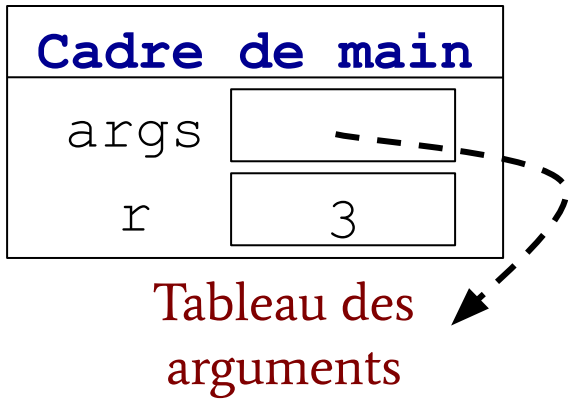
- À la sortie de `add`
  - Reprend l'exécution dans l'appelant avec le résultat



Variables locales (2/2)

- À la sortie de `add`
  - Reprend l'exécution dans l'appelant avec le résultat

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

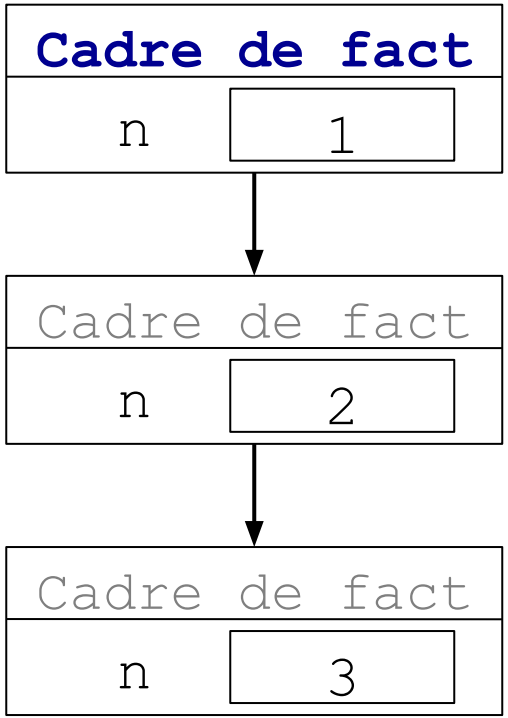


Variables locales et appels récursifs

- Si une méthode s'appelle elle-même
  - Un nouveau cadre à chaque appel

    ⇒ de nouvelles variables locales à chaque appel

```
static int fact(int n) {
  if(n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
/* fact(3) appelle fact(2)
   qui appelle fact(1) */
```



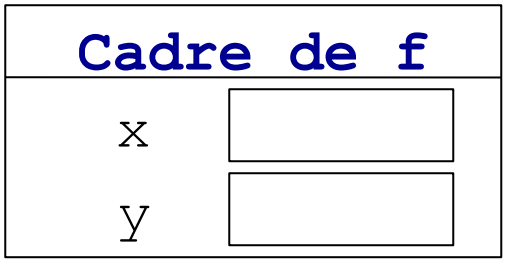
Passage par valeur de types primitifs et de références

- Le passage des arguments se fait **par valeur** : la méthode reçoit une copie de la variable originale.
- On distingue deux cas. La variable est soit :
  - **Un type primitif** : l'appelé reçoit une copie d'une valeur
    - la copie et l'originale **sont différentes**
  - **Une référence** : l'appelée reçoit une copie de la référence vers une valeur
    - la copie et l'originale **sont différentes** mais…
    - la valeur est **partagée** entre l'appelant et l'appelé (à travers la référence)

- En Java :
  - On a 8 types primitifs
    - (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` et `double`)
  - Référence pour les autres types
    - (`String` et tableaux pour l'instant)

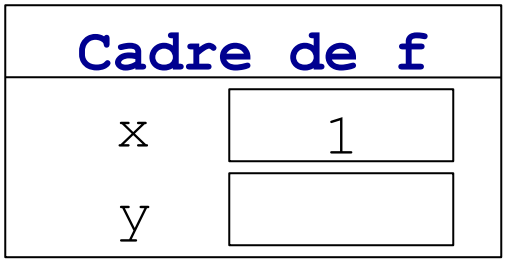
Passage par valeur de types primitifs – exemple

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



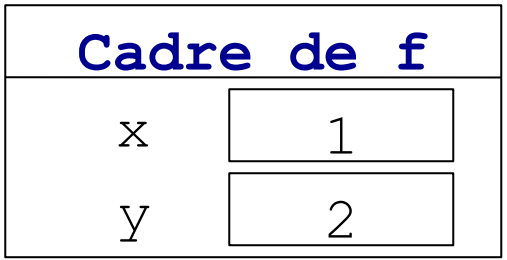
Passage par valeur de types primitifs – exemple

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



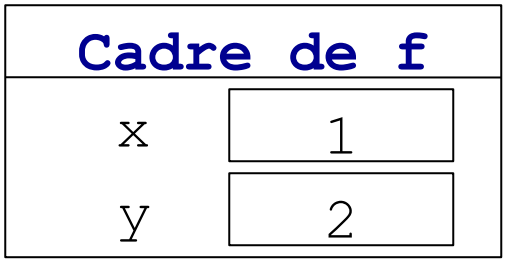
Passage par valeur de types primitifs – exemple

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur de types primitifs – exemple

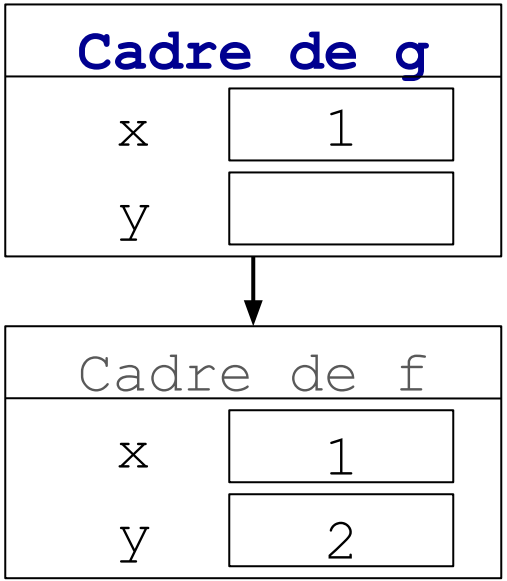
```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur de types primitifs – exemple

- À l'entrée dans g
  - Le `x` du cadre de `f` **est copié** dans le `x` du cadre de `g`

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur de types primitifs – exemple

Attention !

Le fait que les variables de f et g aient
le même nom n'a aucune influence sur l'exécution

Si les variables de g se nommaient a et b,
le programme fonctionnerait exactement de la même façon !

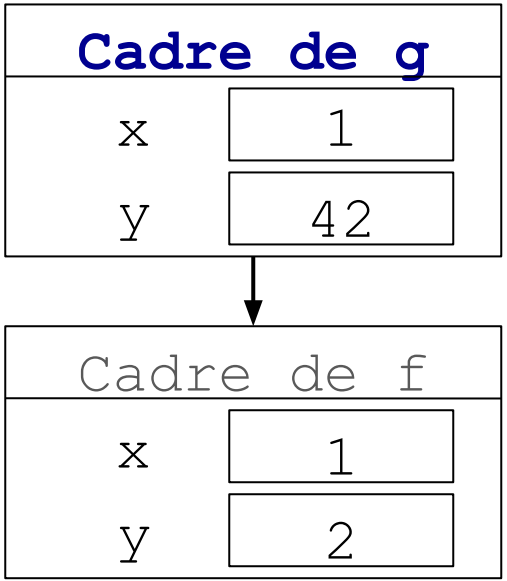
```
  g(x);
}
```

x	1
y	2

Passage par valeur de types primitifs – exemple

- L'affectation du `y` de `g` ne change pas la valeur du `y` de `f`

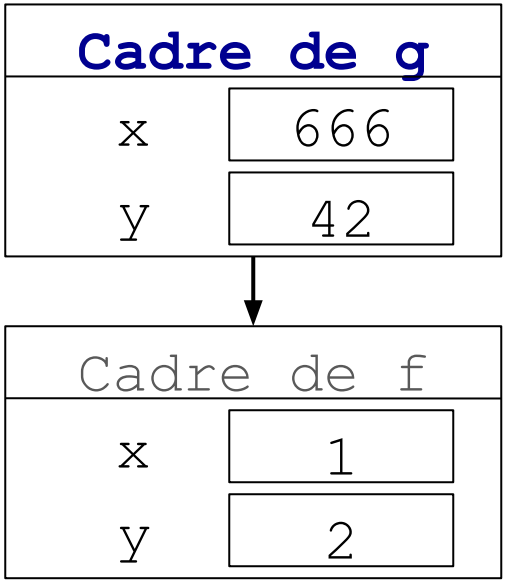
```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur de types primitifs – exemple

- `g` modifie la copie de la variable `x` de `f`, pas celle de `f`

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



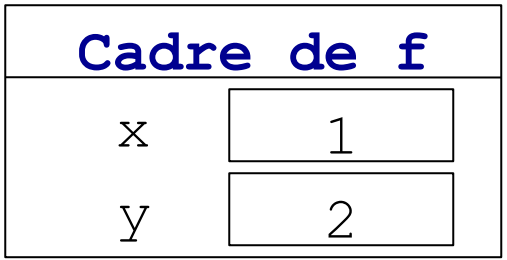
Passage par valeur de types primitifs – exemple

Pour résumer :

Les variables de l'appelant

ne sont **jamais** modifiées par l'appelé

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



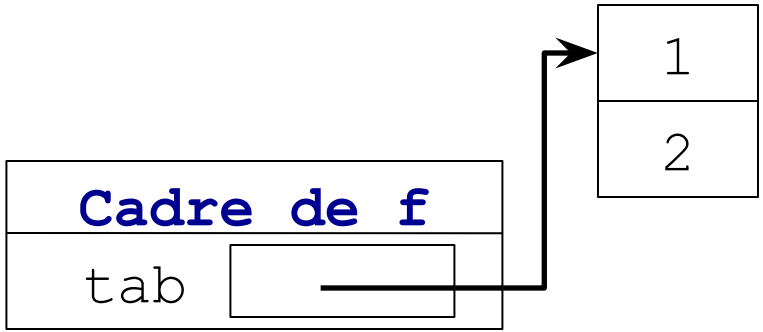
Passage par valeur de références – exemple

```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```

Passage par valeur de références – exemple

- Rappel :
  - Un tableau est alloué dans la mémoire
  - La variable `tab` contient une **référence** vers ce tableau

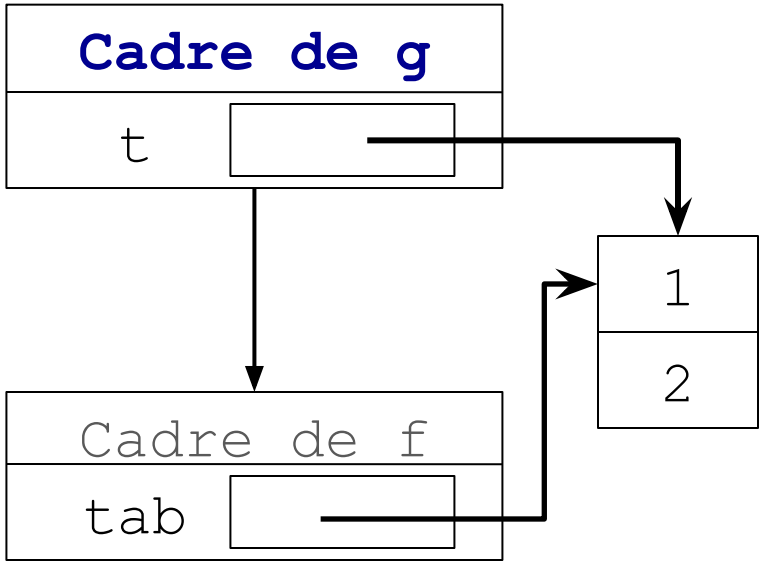
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par valeur de références – exemple

- À l'entrée de g
  - `t` reçoit une copie de `tab`
  - `tab` étant une référence, **`t` et `tab` référence le même tableau**
    - le tableau est passé par référence

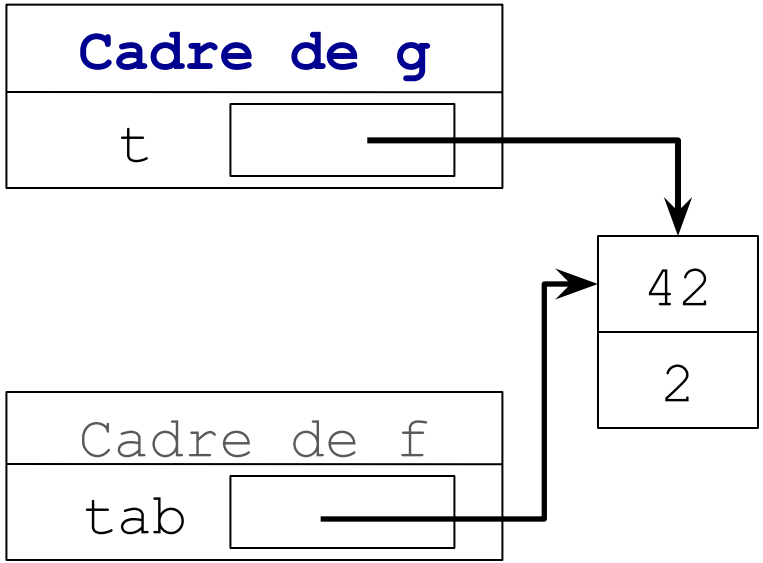
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par valeur de références – exemple

- La modification dans `g` modifie le tableau référencé par `tab`

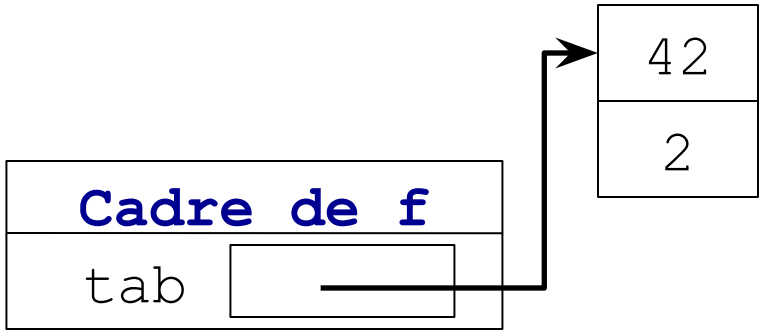
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par valeur de références – exemple

- L'appelant peut donc modifier une valeur à travers une référence

```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par valeur

Pour résumer :

Les variables de l'appelant

ne sont **jamais** modifiées par l'appelé

En revanche, les valeurs **référencées**

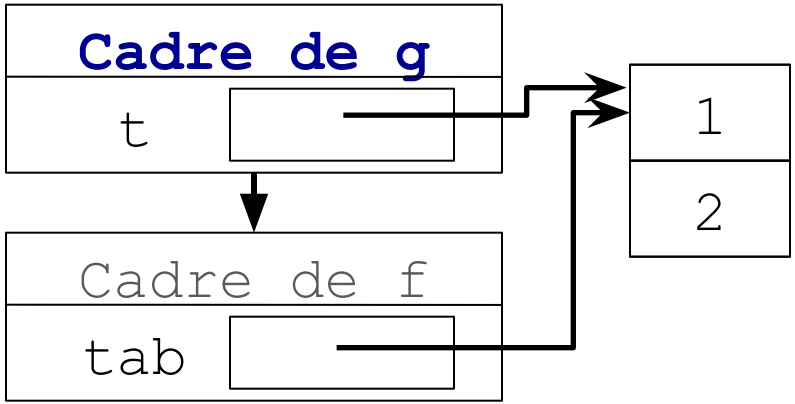
à la fois par l'appelant et l'appelé

peuvent être modifiées par l'appelé

Piège !

- Attention, un tableau est une référence, et la référence est passée **par valeur** (c'est donc une copie)…

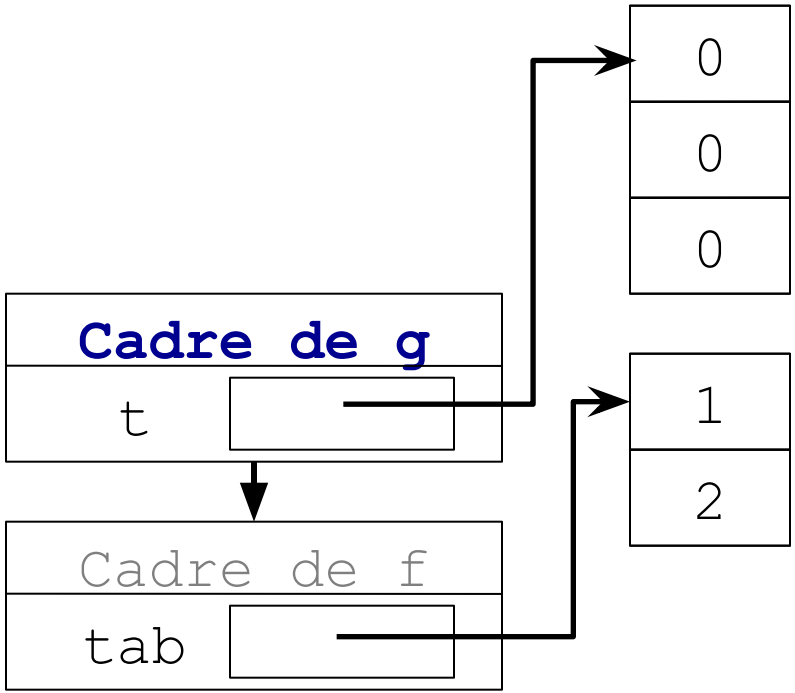
```
static void g(int[] t) {
  t = new int[3];
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
  System.out.println(tab[0]);
}
```



Piège !

- Attention, un tableau est une référence, et la référence est passée **par valeur** (c'est donc une copie)…

```
static void g(int[] t) {
  t = new int[3];
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
  System.out.println(tab[0]);
}
```



Piège !

- Attention, un tableau est une référence, et la référence est passée **par valeur** (c'est donc une copie)…

```
static void g(int[] t) {
  t = new int[3];
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
  System.out.println(tab[0]);
}
```