
# Cours1
# Récursivité, JavaDoc, Tests unitaires

Année 2022-2023 J-F. Kamp - IUT de Vannes

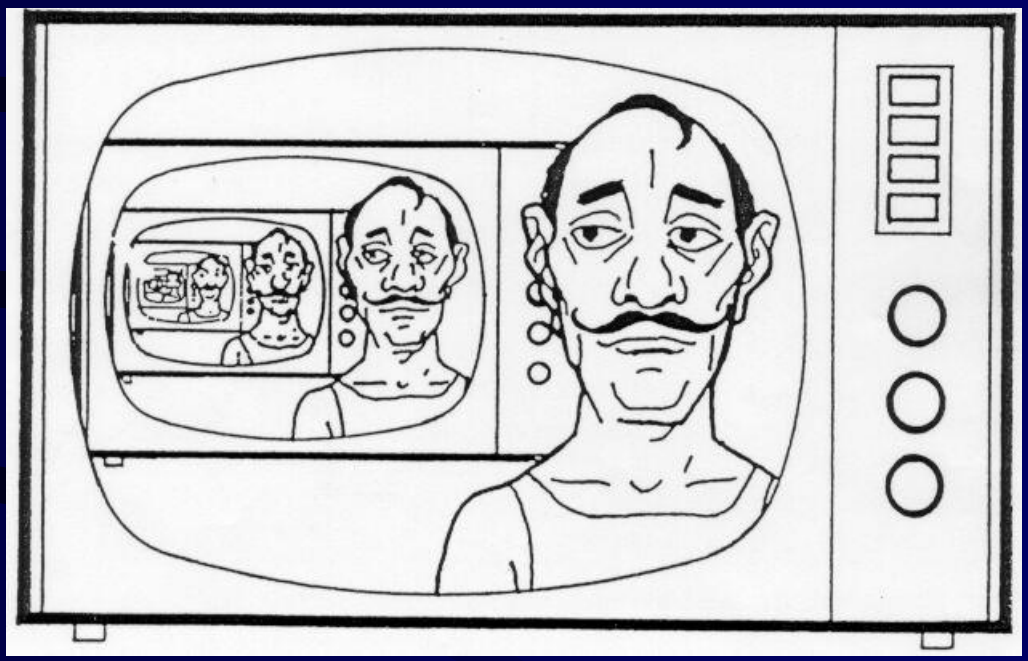
# PLAN

- Récursivité
  - Définition
  - Un exemple simple
  - Mémoire et pile d'appel
  - Intérêt de la récursivité
- JavaDoc
- Tests unitaires
  - Structure générale d'un programme en R1.01.P2
  - Tests unitaires de méthodes

# Récursivité

# Définition

Un objet est récursif s'il s'utilise lui-même dans sa composition ou sa définition.



# Définition

Récursivité en mathématique

- Somme des N premiers entiers positifs :
  - $\sum N = \sum ( N - 1 ) + N$
  - $\sum 0 = 0$
- Puissance N (>=0) d'un nombre X :
  - $X^N = X^{(N-1)} * X$
  - $X^0 = 1$

On peut écrire

- Somme : $F( N ) = F( N-1 ) + N$

  et $F(0) = 0$
- Puissance :

  $F( X, N ) = F( X, (N-1) ) * X$

  et $F( X, 0 ) = 1$

# Définition

Un algorithme récursif P se compose d'un ensemble d'instructions S (ne contenant pas P) et de P **lui-même**.

En Java :

Une méthode est récursive si elle est composée d'instructions dont au moins 1 d'entre elles est la méthode **elle-même**.

# Un exemple simple

Somme des N premiers entiers positifs sous forme récursive :

somEntiers( N ) = somEntiers( N-1 ) + N

```
int somEntiers ( int n ) {

    // variable locale
    int somLoc ;

    somLoc = somEntiers( n-1 ) + n ;

    return somLoc;

}
```

Le fait que la méthode s'appelle elle-même implique **forcément** la mise en place d'une **boucle** qui **ne s'arrêtera PAS** SAUF si on écrit une condition d'arrêt.

# Un exemple simple

Condition d'arrêt : si n égale zéro alors somEntiers(0) = 0 => le sous-programme ne doit plus s'appeler lui-même et la récursivité **doit s'arrêter**.

```
int somEntiers ( int n ) {

    // variable locale
    int somLoc ;

    if ( n = = 0 ) {

        somLoc = 0;   // Arrêt de la récursivité !
    }

    else {

        somLoc = somEntiers ( n-1 ) + n ;

    }

    return somLoc ;

}
```

# Un exemple simple

Somme des N premiers entiers positifs sous forme **itérative** :

```
int somEntiers ( int n ) {

    // variables locales
    int somLoc, i ;

    i = 1 ;
    somLoc = 0;

    // boucle AVEC condition d'arrêt
    while ( i < = n ) {

        somLoc = somLoc + i ;
        i++ ;

    }

    return somLoc ;

}
```

# Mémoire et pile d'appel

Comment cela se passe-t-il en mémoire pour la méthode « somEntiers » ?

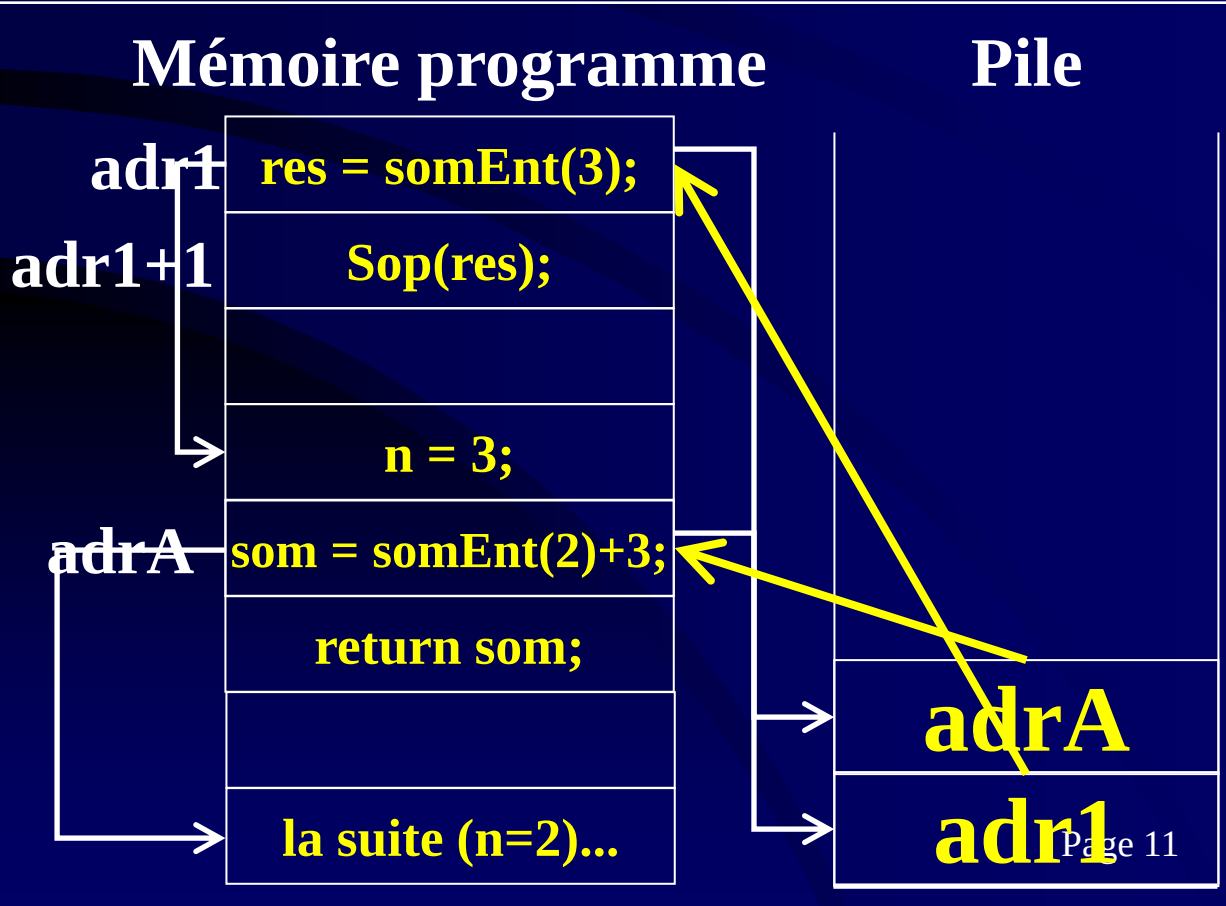
Chaque fois que le processeur rencontre le mot-clé « return » il doit savoir :

- Dans quelle variable il recopie le contenu de « somLoc » ?
- Quelle instruction de quel sous-programme il doit exécuter après ?

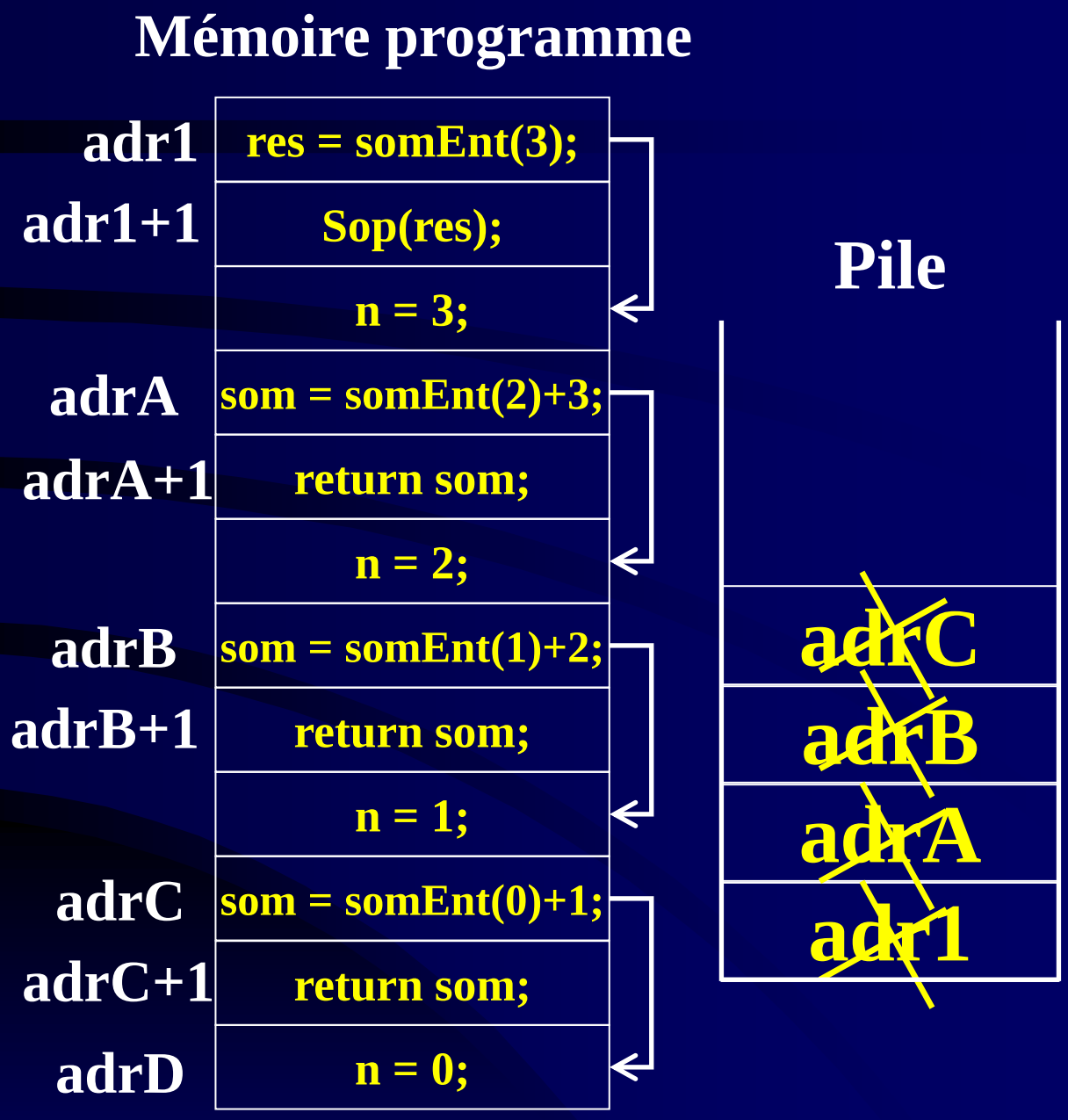
Ces informations sont stockées dans une pile.

# Mémoire et pile d'appel

```
void principal () {
    …
    res = somEnt ( 3 ) ;
    System.out.println ( res ) ;
}

int somEnt ( int n ) {

    // variable locale
    int som;

    som = somEnt( n-1 ) + n ;
    return som;

}
```



# Mémoire et pile d'appel



# Intérêt de la récursivité

Il est toujours possible de transformer une itération en une solution récursive et réciproquement, mais :

- Ce n’est pas toujours évident.
- Chaque solution a ses avantages et ses inconvénients.

Inconvénients de la récursivité ?

- La mémoire consommée est beaucoup plus importante que la version itérative.
- Le temps d’exécution peut être long.
- Estimation difficile de la profondeur maximale de la récursivité.

# JavaDoc

# Java : JavaDoc

Le commentaire dans un code Java est de deux types :

- Le commentaire qui explique un détail d'implémentation

```
// ces lignes sont mises en commentaire
// et ne seront pas compilées
```

```
/* ces lignes sont mises en commentaire
   et ne seront pas compilées non plus
   etc… etc… etc… etc… etc… etc…
   etc… etc… etc… etc… etc… etc… */
```

- Les commentaires de documentation embarquée JavaDoc. C'est ce commentaire là qui sera extrait du source Java pour être transformé en page HTML.

```
/** ces lignes sont destinées à compléter
    le code d'une documentation qui
    explique à quoi sert la classe, la
    méthode etc… */
```

# Java : JavaDoc

Le commentaire JavaDoc peut documenter une classe selon 3 niveaux :

- pour commenter le rôle général de la classe

```
/**
 * Cette classe effectue des opérations élémentaires..
 * etc.
 * etc.
 */

class SimplesTableau {
        ...
}
```

- pour commenter le rôle d'un attribut (variable globale) de la classe

```
class SimplesTableau {

        /** La taille par défaut d'un tableau */
        final int TAILLE = 50;

        ...
}
```

# Java : JavaDoc

- pour commenter le rôle d'une méthode de la classe

```
class SimplesTableau {
        ...

      /**
       * Affiche le contenu des nbElem cases d'un
       * tableau une par une.
       */
      void afficherTab ( int[] leTab, int nbElem ) {
            ...
      }
}
```

A l'intérieur d'un commentaire JavaDoc, toutes les balises HTML standards (<P>, <STRONG>, <B>, <I>, …) de mise en forme sont acceptées.

# Java : JavaDoc

Pour documenter le rôle d'une classe, javaDoc reconnaît un certain nombre de balises (tags) qui lui sont propres :

@author + « l'auteur de la classe »
@version + « numéro de version de la classe »
@since + « version du JDK »
@see + « See Also / référence à quelque chose »

En reconnaissant une balise, JavaDoc choisira le format approprié pour afficher les informations.

```
/**
 * Cette classe effectue des opérations élémentaires..
 *
 * @author J-F. Kamp – octobre 2016
 * @version 1.1.0
 */

class SimplesTableau {

        ...
}
```

# Java : JavaDoc

Pour commenter une méthode, des balises JavaDoc spécifiques sont définies :

@param + « nomParamètre » + « description du paramètre de la méthode » (une balise/paramètre)

@return « description du type retourné » (au maximum un seul type retourné)

```
/**
 * Affiche le contenu des nbElem cases d'un tableau
 * une par une.
 *
 * @param leTab le tableau à afficher
 *
 * @param nbElem le nombre d'entiers que contient le
 * tableau
 *
 */

void afficherTab ( int[] leTab, int nbElem ) { . . . }
```

# Java : JavaDoc

Pour générer les pages HTML de documentation d'une classe, taper la commande :

```
javadoc [options] ../src/MyClass.java
```

```
[options] :
    -d cheminRépertoireHTML
    -version
    -author
    -private
    -...
```

Pour les autres options consulter le site de ORACLE
http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

# JavaDoc : Format des pages

1. La description de la classe

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

## Class Duree

```
java.lang.Object
  |
  +--Duree
```

public class **Duree**
extends java.lang.Object

**Description**

Le type "Duree" permet la manipulation d'intervalles de temps : conversion en JHMS, addition de durées, comparaison entre durées.

# JavaDoc : Format des pages

2. La description résumée des attributs et méthodes

## Field Summary

| | |
|---|---|
| private long | **leTemps**<br>la durée s'exprime en millisecondes |

## Constructor Summary

| |
|---|
| **Duree**(Duree autreDuree)<br>Constructeur qui clone une Duree existante. |
| **Duree**(int heures, int minutes, int secondes)<br>Constructeur de Duree avec initialisation. |
| **Duree**(long millisec)<br>Constructeur de Duree avec initialisation. |

## Method Summary

| | |
|---|---|
| void | **ajoute**(Duree uneDuree)<br>Modificateur qui ajoute une durée à la durée courante. |
| int | **compareA**(Duree autreDuree)<br>Effectue une comparaison entre la durée courante et une autre durée. |
| private int[] | **enJHMS**()<br>Effectue un découpage de la durée en intervalles (jours, heures, minutes, secondes, millisecondes). |
| java.lang.String | **enTexte**(char mode)<br>Renvoie une image textuelle de la durée courante. |
| long | **getLeTemps**()<br>Accesseur qui retourne la valeur de la durée en millisecondes. |

# JavaDoc : Format des pages

## 3. La description détaillée des attributs et méthodes

### Field Detail

**leTemps**

```
private long leTemps
```

la durée s'exprime en millisecondes

### Constructor Detail

**Duree**

```
public Duree(Duree autreDuree)
```

Constructeur qui clone une Duree existante.

**Parameters:**

`autreDuree` - autre durée à copier

---

**Duree**

```
public Duree(int heures,
             int minutes,
             int secondes)
```

Constructeur de Duree avec initialisation.

**Parameters:**

`heures` - nombre d'heures

`minutes` - nombre de minutes

`secondes` - nombre de secondes

Initialisation d'une durée par un nombre d'heures, minutes et secondes.

---

# Tests unitaires

# Structure générale

- Découpage en méthodes

  => void afficherTab (int[] tab, int nb)

  => rôle bien identifié

- Chaque méthode est testée individuellement (test unitaire)

```
void testAfficherTab() {
    int[] monTab = new int[10];
    // test cas normal
    afficherTab ( monTab, 10 );
    // autres tests
    …
}
```

- Le principal appelle les cas de test

```
void principal() {
    testAfficherTab();
    …
}
```

# Structure générale

```
public class MonAlgo {
    // Déclaration des CONSTANTES
    final int TAILLE = 50 ;

    void principal() {
        testMaMeth1();
        …
    }

    void testMaMeth1() {
        // déroulement du test de maMeth1(…)
        …
    }

    int maMeth1 ( int[] tab, double d ) {
        // déclaration des variables locales
        int ret;
        …
        …
        return ret;
    }

}
```

# Tests unitaires de méthodes

Un exemple concret de méthode Java

```
/**
   *  Cette méthode calcule la somme
   *  des éléments contenu dans le
   *  tableau passé en paramètre
   *  @param leTab le tableau des valeurs
   *  @param nbElem le nbre de valeurs
   *  @return la somme des valeurs
   */
int somTab ( int [ ] leTab, int nbElem ) {
   int som, i ;    // variables locales
   som = 0 ;
   for ( i = 0; i < nbElem; i++ ) {
      som = som + leTab[i] ;
   }
   return som ;
}
```

# Tests unitaires de méthodes

Test de la méthode dans 3 familles de cas distincts :

- Cas normaux

  Examiner le comportement de la méthode sur des données standards et vérifier la validité des résultats.
- Cas limites

  Comportement de la méthode sur des données à la limite de l'acceptable et vérifier la validité des résultats.
- Cas d'erreurs

  Comportement de la méthode sur des données qui rendent son exécution impossible : la méthode doit afficher un message d'erreur.

# Tests unitaires de méthodes

Test de la méthode **int somTab ( … )** dans un cas normal :

```
void testSomTab () {

    // variables locales
    int som;

    Sop ( "==Test de la méthode somTab==" );

    Sop ( "Cas normal : un tableau avec quelques entiers" );

    int[] tab1 = { -2, -3, 45, 80 };

    som = somTab ( tab1, 4 );

    // automatiser les tests le + possible
    if ( som == 120 ) Sop ( "Test réussi" );
    else Sop ( "Echec du test" );

    …
```

# Tests unitaires de méthodes

Test de la méthode **int somTab ( … )** dans un cas limite :

```
void testSomTab () {

    …

    Sop ( "Cas limite : un tableau avec zéro entiers" );

    int[] tab2 = {};

    som = somTab ( tab2, 0 );

    // automatiser les tests le + possible
    if ( som == 0 ) Sop ( "Test réussi" );
    else Sop ( "Echec du test" );

    …
```

# Tests unitaires de méthodes

Test de la méthode **int somTab ( … )** dans un premier cas d'erreur :

```
void testSomTab () {

    …

    Sop ( "Cas d'erreur1 : un tableau inexistant" );

    int[] tab3 = null;

    Sop ( "Message d'erreur attendu" );

    som = somTab ( tab3, 0 );

    …
```

# Tests unitaires de méthodes

Test de la méthode **int somTab ( … )** dans un second cas d'erreur :

```
void testSomTab () {

    …

    Sop ( "Cas d'erreur2 : un tableau avec un nbre d'entiers incohérent" );

    int[] tab4 = { -66, 0, 102 };

    Sop ( "Message d'erreur attendu" );

    som = somTab ( tab4, 6 );

    Sop ( "==Fin du test de la méthode somTab==" );

}
```