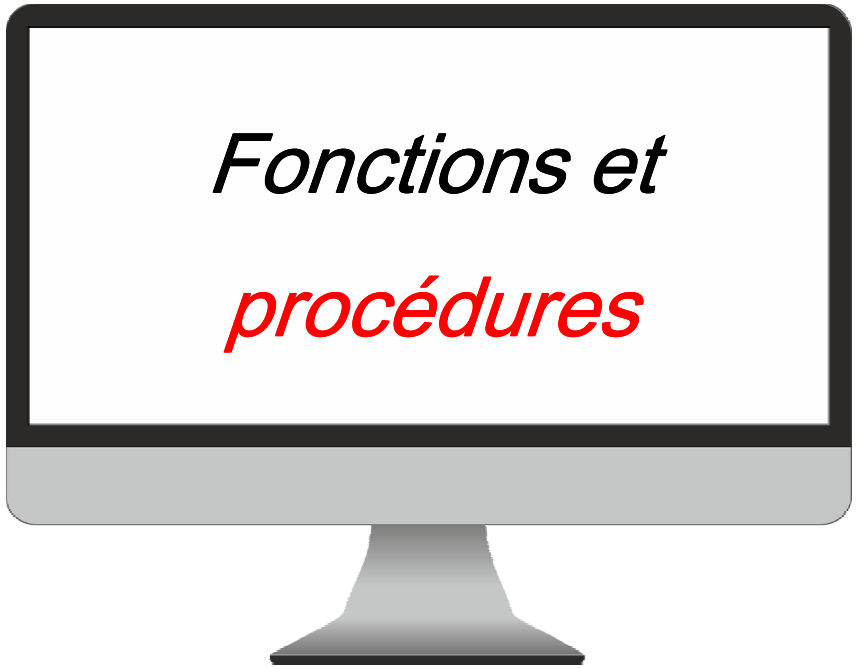
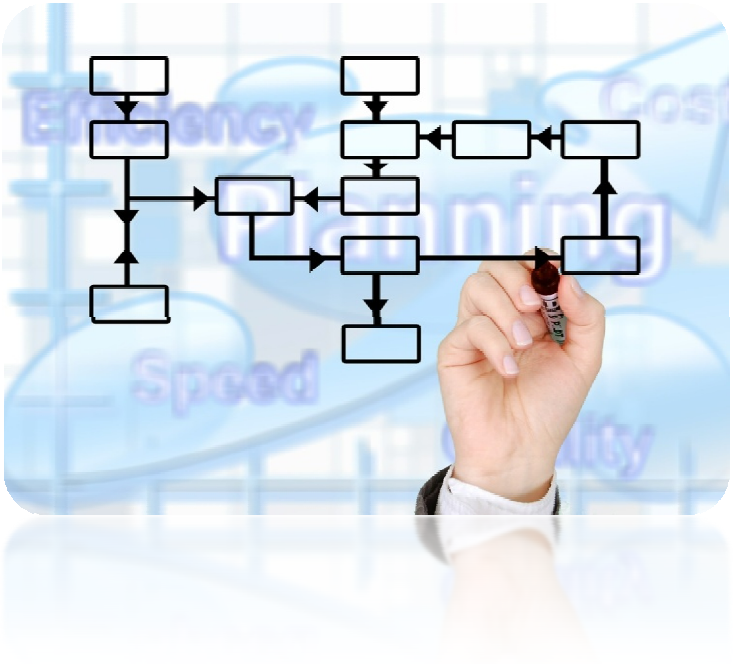


Fonctions et procédures

Objectifs : apprendre à modulariser les traitements sous forme de fonctions et procédures.



1- Module, fonction, procédure

2- La syntaxe des procédures

3- La syntaxe des fonctions

Modularité = diviser pour mieux régner

1 La modularité est essentielle

Un programme modulaire est constitué de plusieurs parties (= modules). Modulariser un programme permet de développer plus rapidement en factorisant du code répétitif.

- Factoriser le code

  Si des tâches analogues sont nécessaires à plusieurs endroits de votre programme (même traitement exécuté, mais avec des données « paramètres » ou «arguments » différentes), il faut factoriser le code.

- La factorisation donne une organisation efficace
  - analyse du détail à l'ensemble.
  - réutilisation = écrit 1 fois, utilisé +sieurs fois
  - bibliothèque = ensemble de modules
  - les modules peuvent être codés par plusieurs personnes.

- En POO, la factorisation est systématique

Modularité = diviser pour mieux régner

2 Qu'est-ce qu'un module ?

Un module = partie d'un algorithme (ou d'un programme).
On l'identifie (= on lui donne un nom) afin d'appeler plutard son exécution.

Il peut s'agir :

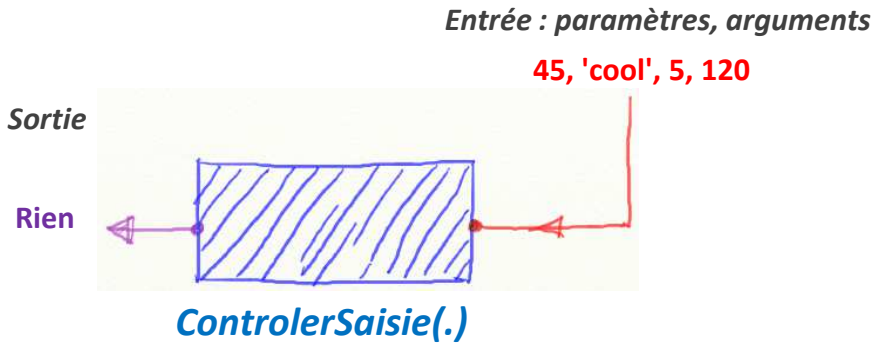
- D'une fonction
- D'une procédure
- Ou simplement d'un bloc de code comme en langage Python (selon la technologie, la signification du mot module diffère).

Modularité = diviser pour mieux régner

1 Définition d'une procédure

Une procédure est un ensemble d'instructions regroupées sous un nom, qui réalise un traitement particulier dans un programme lorsqu'on l'appelle.

Elle ne retourne **aucune** valeur (pas de clause RETOURNE) mais **peut modifier une ou plusieurs données** (*fournies en paramètres*) de l'application. ***Son appel équivaut à une nouvelle instruction.***



Exemple : ***ControlerSaisie(nombre, msg, min, max)***

Modularité = diviser pour mieux régner

2 La syntaxe d'une procédure

La procédure doit porter un nom.
Elle peut déclarer ses propres données, et avoir son propre bloc de traitement. Mais contrairement à un programme, elle ne peut pas s'exécuter indépendamment d'un autre programme.

Paramètres formels

```
PROCEDURE nom(déclaration des paramètres)

VAR
      (* ici, définition des variables locales *)
DEBUT
   (* Ici, les différents traitements *)

FIN-PROC
```

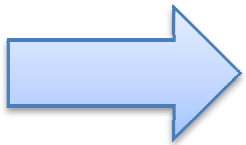
Variables valables uniquement dans la procédure

Modularité = diviser pour mieux régner

4 Procédure sans paramètre et son appel

```
PROCEDURE ligneEtoile()
VAR  i : entier
DEBUT
        POUR i DE 1 A 10 FAIRE
            ECRIRE '*'
        FIN-POUR

        ECRIRE ' \n'
FIN-PROC
```

Affiche une étoile et va à la ligne



```
ALGO RectangleEtoile
VAR  nlignes, cpt : entier
DEBUT
    ECRIRE "Combien voulez-vous d'étoiles?"
    LIRE nlignes

    POUR cpt DE 1 A nlignes FAIRE
        ligneEtoile()
    FIN-POUR
FIN
```

Programme dessinant un rectangle d'étoiles

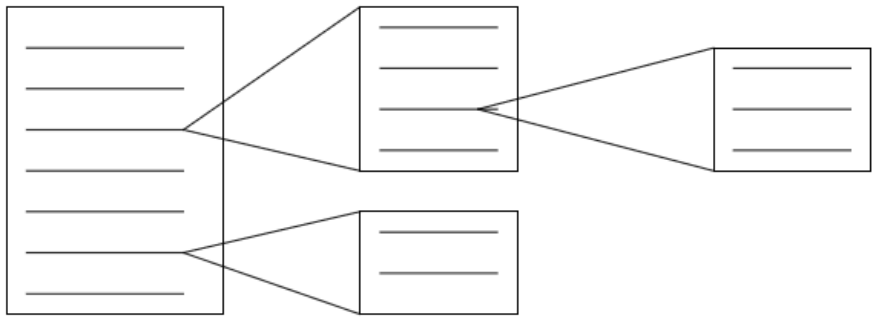
```
VAR  i : entier
DEBUT
    POUR i DE 1 A 10 FAIRE
        ECRIRE '*'
    FIN-POUR

    ECRIRE ' \n'
FIN-PROC
```

Modularité = diviser pour mieux régner

5 Principe des appels

Lorsque le processeur rencontre l'appel d'une procédure, il arrête momentanément l'exécution du programme appelant pour aller exécuter les instructions de la procédure. Quand il a terminé l'exécution de la procédure, le processeur reprend l'exécution du programme appelant là où il s'était arrêté.

Une procédure peut être appelée soit par un programme, soit par un autre sous-programme (qui lui même a été appelé). Les appels de sous-programmes peuvent s'imbriquer autant qu'on le désire.



Imbrication des appels

Modularité = diviser pour mieux régner

6 Procédure avec paramètre et appel

Paramètre formel

```
PROCEDURE ligneEtoile(donnée nombre: entier)
VAR  cpt : entier
DEBUT
        POUR cpt DE 1 A nombre FAIRE
            ECRIRE '*'
        FIN-POUR

        ECRIRE ' \n'
FIN-PROC
```

Affiche des étoiles

Appel avec un paramètre réel

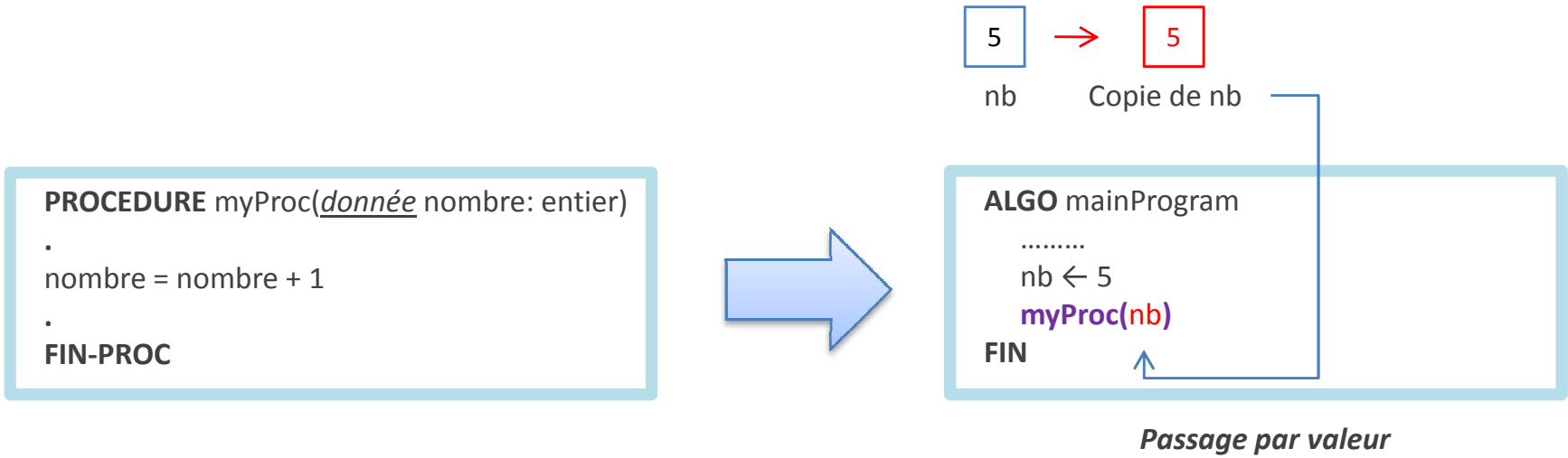
```
ALGO RectangleEtoile
VAR  nlignes, netoiles, i : entier
DEBUT
    ECRIRE "Combien d'étoiles par ligne?"
    LIRE netoiles

    ECRIRE "Combien de lignes?"
    LIRE nlignes

    POUR i DE 1 A nlignes FAIRE
        ligneEtoile(netoiles)
    FIN-POUR
FIN
```

Programme dessinant un rectangle d'étoiles

Modularité = diviser pour mieux régner

7 Passage de paramètres : par valeur (= copie)

Dans ce cas on passe à la procédure appelée **myProc(.)** le paramètre réel **nb** (en rouge ici) qui est une copie de la variable **nb** (en noir). A la fin de l'exécution de **myProc(.)** , **nb** contient toujours la même chose.

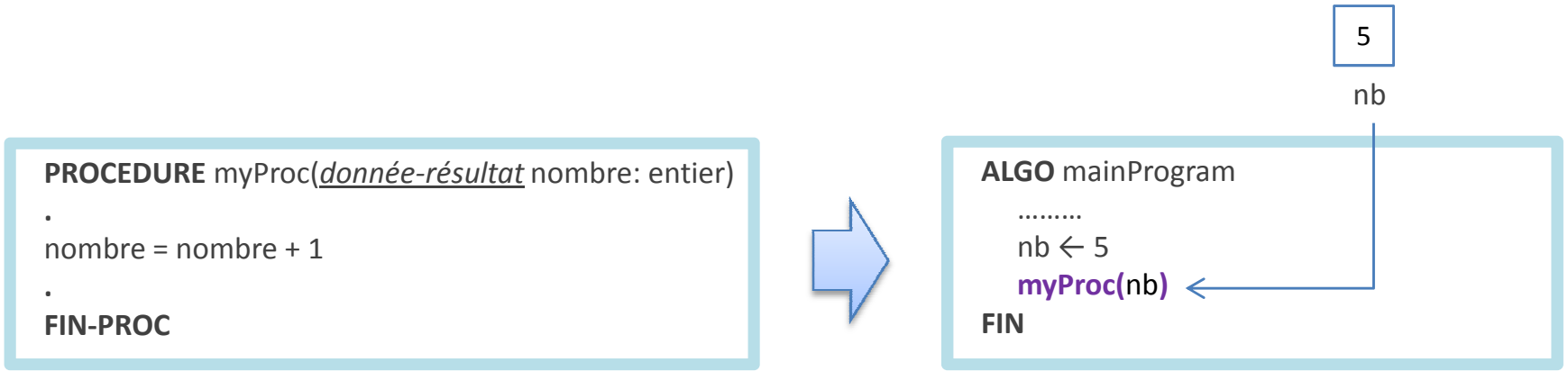


Passage par valeur

Modularité = diviser pour mieux régner

8 Passage de paramètres : par référence (= pointeur = adresse)

Dans ce cas on passe à la procédure appelée **myProc(.)** l'adresse de la variable **nb**. On ne fait pas de copie de valeur. A la fin de l'exécution de **myProc(.)** , **nb** ne contient plus la même chose.



Passage par adresse

Modularité = diviser pour mieux régner

9 Traduction en langage Java

Prenons un exemple très simple d'une procédure **direAge(.)** qui affiche juste l'âge qu'on passe en paramètre. Créer le fichier **TestAlgo.java** ci-dessous, compilez puis exécutez-le.

```
PROCEDURE direAge(donnée age : entier)
DEBUT
        ECRIRE 'Votre âge est', age
FIN-PROC
```

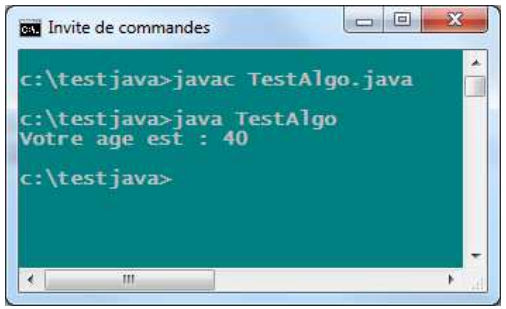
Procédure

Rien n'est renvoyé (**void**)

```
public class TestAlgo {
    public static void main(String[] args) {
        // Donnée à passer à la procédure
        int monAge = 40;

        // Appel de la procédure
        direAge(monAge);
    }

    // La procédure (méthode)
    public static void direAge(int age) {
        System.out.println("Votre age est : " + age);
    }
}
```

Appel avec un argument entier

Fichier TestAlgo.java

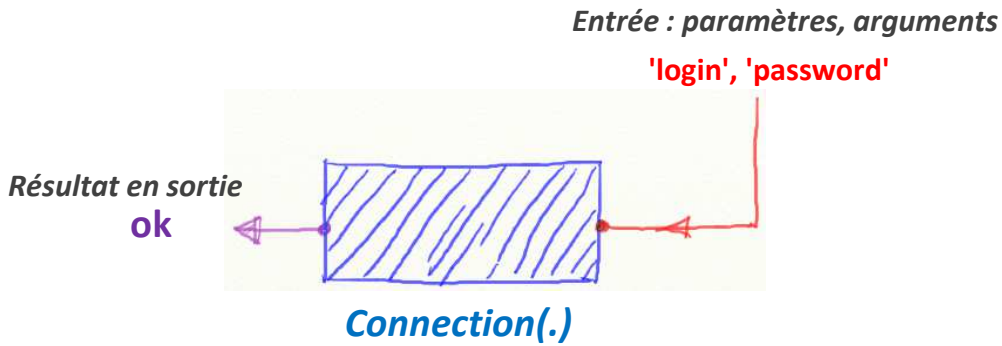


Compilation et exécution

Modularité = diviser pour mieux régner

1 Définition d'une fonction

Une fonction nomme une **valeur** résultant d'un calcul (arithmétique, booléen…). Elle agit comme une boîte noire qui reçoit en entrée des données ou rien, et ***retourne un (et un seul)*** résultat en sortie. Elle équivaut donc à une opération : son appel peut figurer dans une expression.



Exemple : ok ← ***Connection(***'login', 'password'***)***

Modularité = diviser pour mieux régner

2 La syntaxe d'une fonction

La fonction doit porter un nom.
Elle peut déclarer ses propres données, et avoir son propre bloc de traitement.

Paramètres formels

FONCTION nom(déclaration des paramètres) : type de la valeur retournée

VAR
(* ici, définition des variables locales *) ← Variables valables uniquement dans la fonction

DEBUT
(* Ici, les différents traitements *)
RETOURNE valeur à retourner ← instruction obligatoire : retourne la valeur au programme appelant.
FIN-FONCTION

Correspond à la valeur d'une variable, d'une expression ou d'une valeur littérale.

Modularité = diviser pour mieux régner

3 Appeler une fonction

Le ***programme appelant*** est celui qui utilise la fonction.

L'appel d'une fonction doit obligatoirement se trouver à l'intérieur d'une instruction (affichage, affectation, et.) qui utilise sa valeur.

A l'appel on doit fournir les valeurs (= ***arguments = paramètres réels***) des paramètres de la fonction : ici, **login** et **password**.

```
ALGO Lecture

VAR  login, password : CHAINE
       ok : BOOLEEN

DEBUT

   ECRIRE "Entrez l'identifiant et le mot de passe"
   LIRE login, password

   (* Appel de la fonction Connnexion(.) *)
   ok ← Connection(login, password)

FIN
```

Le programme appelant (exemple)

Modularité = diviser pour mieux régner

4 Exemple : définition de 2 fonctions

```
FONCTION factorielle(n: entier) : entier
VAR  i, fact : entier
DEBUT
    fact ← 1

    SI n ≠ 0
        POUR i DE 1 A n FAIRE
            fact ← fact * n
        FIN-POUR
    FIN-SI

    RETOURNE fact
FIN-FONCTION
```

fonction qui calcule la factorielle du nombre passé en paramètre.

```
FONCTION saisie() : entier
VAR  nb : entier
DEBUT
    ECRIRE 'Veuillez entrer un nombre positif'
    LIRE nb

    TANT QUE nb < 0
        ECRIRE 'Erreur. Saisissez un entier > 0'
        LIRE nb
    FIN-TANT-QUE

    RETOURNE nb
FIN-FONCTION
```

fonction sans paramètre qui permet de saisir un nombre positif qui est retourné

Modularité = diviser pour mieux régner

5 Exemple : utiliser les 2 fonctions précédentes

Les paramètres d'une fonction sont typés. La valeur des arguments (=paramètres effectifs) à l'appel est recopiée dans les paramètres formels qui servent à réaliser le traitement de la fonction.

```
ALGO afficheFactorielle
VAR  nombre : entier, rep : chaine
DEBUT

   REPETER
      nombre ← saisie()
      ECRIRE 'La factorielle de ', nombre, ' est ', factorielle(nombre)
      ECRIRE 'Voulez-vous recommencer? (oui/non) '
      LIRE rep
   JUSQU'À rep

   ECRIRE 'A bientôt'
FIN
```

Appel de la fonction **saisie()** et affectation de sa valeur retournée à la variable **nombre**

paramètre effectif = argument

Appel de la fonction **factorielle(.)** et affichage de sa valeur retournée

Utilisation des 2 fonctions saisie() et factorielle(.)

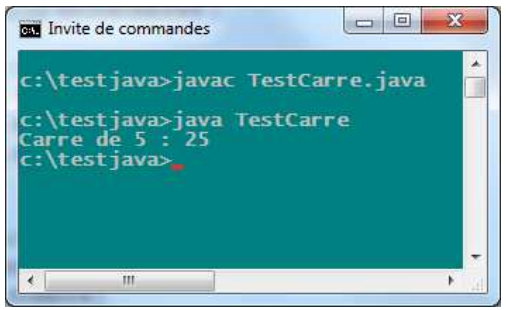
Modularité = diviser pour mieux régner

6 Traduction en langage Java

Prenons un exemple très simple d'une fonction **carre(.)** qui reçoit un entier en paramètre et retourne son carré. Créer le fichier **TestCarre.java** ci-dessous, compilez puis exécutez-le.

```
FONCTION carre(donnée nombre : entier) :entier
DEBUT
        RETOURNE nombre * nombre
FIN-PROC
```

Procédure



Compilation et exécution

Appel ***dans une instruction*** avec un paramètre entier

On renvoie un entier

```java
public class TestCarre {
    public static void main(String[] args) {
        // Donnée à passer à la fonction
        int monNombre = 5;

        // Appel de la fonction
        int leCarre = carre(monNombre);

        // Affichage du carré
        System.out.print("Carre de ");
        System.out.print(monNombre + " : ");
        System.out.print(leCarre);
    }

    // La fonction (méthode)
    public static int carre(int nombre) {
        return nombre * nombre;
    }
}
```

Fichier TestCarre.java