Algorithmique & Programmation

La notion de fonction (2)

yann.secq@univ-lille.fr

ABDELKADER Omar, BIRLOUEZ Martin, BONEVA Iovka, DELECROIX Fabien, LEQUINIOU Erwann, MARSHALL-BRETON Christopher, REKIK Yosra, SECQ Yann, SOW Younoussa, SUDHEENDRAN Megha, SU Yue

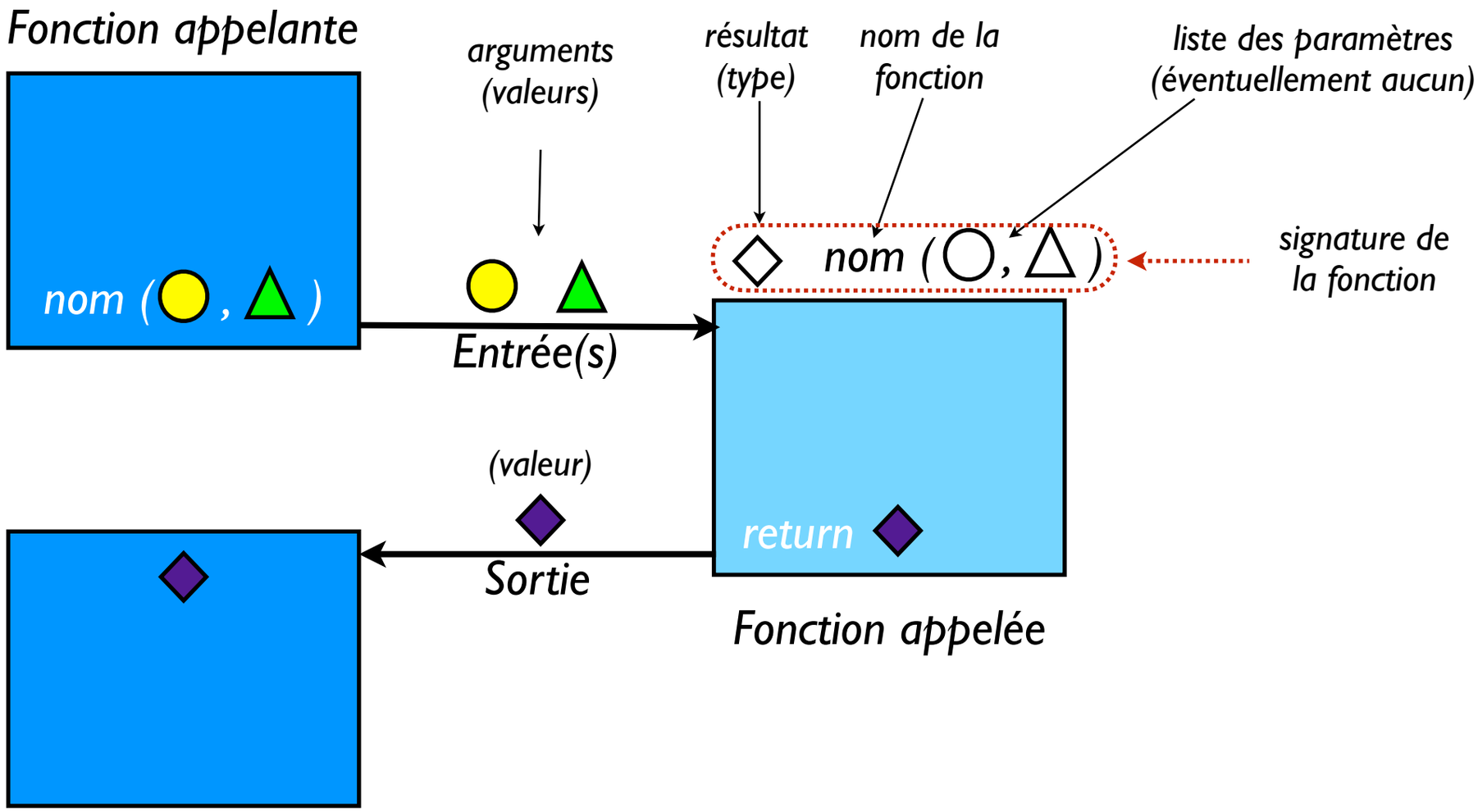
Saison 2025-2026 / R1.01+R1.02
Initiation au développement

Notion de fonction

```
<type de retour> nom(<liste de paramètres>)
```

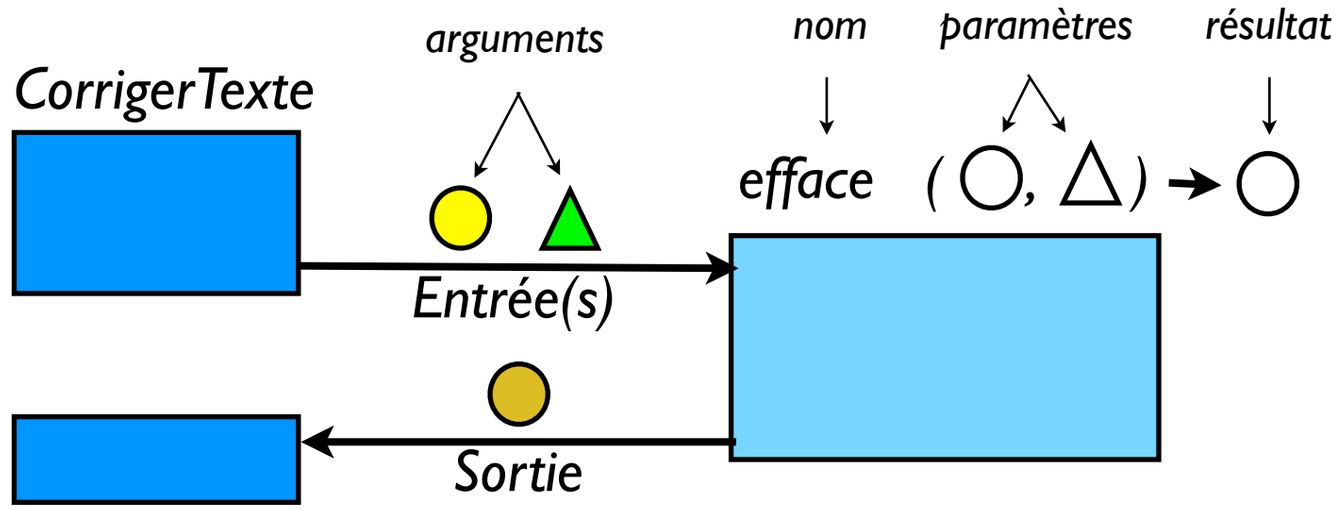
- **Une fonction est définie par sa signature**
  - le **nom** de la fonction (le plus pertinent possible !)
  - les **paramètres** (informations nécessaires pour réaliser le calcul)
  - le **type de son résultat**
- **ATTENTION : un seul `return` par fonction et toujours en dernière instruction !**

Notion d'appel de fonction



Appel d'une fonction

- Lors de l'appel à une fonction, la machine :
  - recherche la fonction correspondante : même nom, même paramètres
  - transmets les **valeurs** (ou arguments) présentes dans l'appel en les associant aux paramètres de la fonction
  - exécute le corps de la fonction
  - lorsque le mot-clé `return` est rencontré, la valeur de l'expression est renvoyée et se substitue l'appel de fonction



Paramètres

`msg` **et** `c`

Arguments

`texte` **et** `'e'`

La valeur contenue dans la variable texte

```
class CorrigerTexte extends Program {

                                  ○       △
  String copieSans(String msg, char c) {
    String resultat = "";
    for (int idx=0; idx<length(msg); idx++) {
      if (charAt(msg,idx) != c) {
        resultat = resultat + charAt(msg,idx);
      }
    }                  ●
    return resultat;
  }
  void algorithm() {
    String texte = readString();
    println(texte);          ●     ▲
    println(copieSans(texte,'e'));
  }                  ▬▬▬▬▬▬▬▬▬▬▬
}                      ●
```

Passage par valeur

- Tous les types “simples” (dits *primitifs*) sont passés par valeur : `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.
- Pour l’instant, laissons `String` avec les types primitifs
- Passer une information par valeur revient à en faire une copie
- On ne peut donc modifier la valeur dans la fonction !

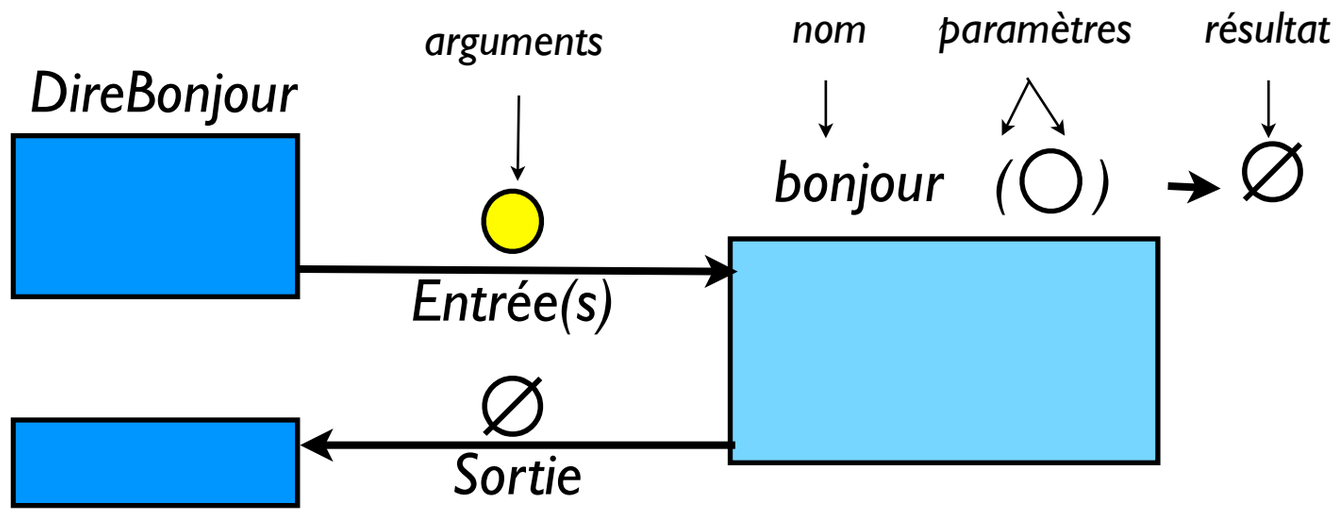
Passage par valeur

```
class PassageValeur extends Program {
  int add1(int op1, int op2) {
    return op1 + op2;
  }
  int add2(int op1, int op2, int res) {
    res = op1 + op2;
    return 0;
  }
  void add3(int op1, int op2, int res)  {
    res = op1 + op2;
  }
  void algorithm() {
    int a = 3, b = 4, resultat = -1;
    println(add1(a, b));
    println(add2(a,b,resultat));
    println(resultat);
    add3(a,b,resultat);
    println(resultat);
  }
}
```

Quels affichages sont produits ?

Fonctions et procédures

- Une fonction retourne toujours une valeur
- Une procédure est une fonction ne retournant pas de valeur ...
- Ou plus précisément, retournant “vide” : `void`
- Une procédure ne peut jamais être utilisée dans une expression (car pas de valeur retournée) !
- C’est le cas de `void algorithm()` ;)




```
class DireBonjour extends Program {
   ∅                              ○
  void bonjour(String nom) {
    println("Bonjour "+nom+" !");
  }

  void algorithm() {
    String patronyme;
    print("Veuillez entrer votre nom: ");
    patronyme = readString();
  ∅ bonjour(patronyme);
  }
}
```

Une procédure ne peut jamais être utilisée dans une expression (car `void` !) !

(Rappel) Notion de portée

- Les variables ont une portée bien définie
- Une variable n'existe que dans le bloc où elle est déclarée (accolades ouvrantes/fermantes entourant sa déclaration)
- Avant, la vie était simple ... (sauf `for` !)
- Maintenant, soyez attentifs à cette notion de portée avec les fonctions !
- **ATTENTION : variables globales à utiliser avec grande modération !**

nom *est une variable* ***globale***

resultat *est une variable* ***locale*** *à* copieSans

i *est une variable* ***locale*** *à la boucle* for

texte *est une variable* ***locale*** *à* algorithme

```
class CorrigerTexte extends Program {
  String nom  = "Turing";
  String copieSans(String msg, char c) {
    String resultat = "";
    for (int i=0; i<length(msg); i=i+1) {
      if (charAt(msg, i) != c) {
        resultat = resultat + charAt(msg, i);
      }
    }
    return resultat;
  }

  void algorithm() {
    String texte;
    texte = readString();
    println(copieSans(texte, 'e'));
    println(copieSans(nom, 'u');
  }
}
```

Surcharge de fonction

- On peut définir des fonctions ayant le même nom, mais **il faut des signatures différentes**
- Exemple: `print` **ou** `println` !
- **Surcharger une fonction signifie créer une nouvelle fonction avec le même nom mais une signature différente**
- Pratique pour des fonctions réalisant un traitement similaire ou avec différents paramétrages

Exemple : `random`

- Soit la fonction `random` qui tire un nombre aléatoirement dans `[0.0, 1.0[`
- Il serait pratique d'avoir des tirages entre `[0, n]` ou encore `[min, max]` avec `n`, `min`, `max` entiers
- Fondamentalement, cela reste un tirage aléatoire …
- Surchargeons la fonction `random` :
  - `double random()` : la fonction prédéfinie qui tire dans `[0.0, 1.0[`
  - `int random(int n)` : un entier tiré aléatoirement dans `[0, n]`
  - `int random(int min, int max)` : un entier tiré dans `[min, max]`

```
class RandomPratique extends Program {

  int random(int borneMin, int borneMax) {
    int alea = (int) (random()*(borneMax-borneMin));
    return borneMin + alea;
  }
  // Surcharge en réutilisant la fonction précédente!
  int random(int n) {
    return random(0, n);
  }

  void algorithm() {
    int min = 1;
    int max = 6;
    println(min+" <= "+random(min, max)+" <= " +max);
    println(min+" <= "+random(max)+" <= "+max);
  }
}
```

Surcharges de la fonction `random()`

Comment s'assurer que nos fonctions sont valides ?

- Exécuter de nombreuses fois le programme, pour vérifier que les différentes valeurs possibles apparaissent bien … un peu long !
- Il serait pratique de pouvoir automatiser cette vérification
- **C'est tout l'intérêt des assertions et fonctions de tests :)**

```
class RandomSurcharge extends Program {
  String copieSans(String phrase, char c) { … }
  int random(int borneMin, int borneMax) { … }
  // La fonction testée (indirectement la précédente aussi)
  int random(int n) { … }
  // Reconnue comme une fonction de test grâce au préfixe !
  void testRandomN() {
    String nombres = "0123456";
    for (int tirage=0; tirage<100000; tirage=tirage+1) {
      nombres = copieSans(nombres, charAt(""+random(6),0));
    }
    assertEquals("" , nombres);
  }

  void algorithm() { … }
}
```

Conversion d'un int en char, version brutale …

Assertion vérifiant que la variable contient bien une chaîne vide

Test de la fonction `random(int n)`

Comment interpréter ces messages ?

```
Summary of RandomPratique Test Results

Your tests: 50% (1/2)
✅ testCopieSans
❌ testRandomN
   Expected:||
   Received:|6|

Professor tests: 0% (0/0)

Total: 50% (1/2 yours | 0/0 professor)
```

La fonction `testCopieSans` n'a pas détecté d'invalidité : toutes ses assertions étaient donc vérifiées :)

La fonction `testRandomN` a détecté une invalidité : une assertion n'a pas été vérifiée et a stoppé l'exécution de cette fonction de test.

On nous informe que l'assertion d'égalité n'a pas été vérifiée et que la chaîne vide n'est pas égale à 6

Pourquoi ?

```
class RandomSurcharge extends Program {

  int random(int borneMin, int borneMax) {
    int alea = (int) (random()*(borneMax-borneMin));
    return borneMin + alea;
  }

  int random(int n) {
    return random(0, n);
  }

  void algorithm() {
    int min = 1;
    int max = 6;
    println(min+" <= "+random(min, max)+" <= " +max);
    println(min+" <= "+random(max)+" <= "+max);
  }
}
```

Où se cache le bug ?

```
class RandomSurcharge extends Program {

  int random(int borneMin, int borneMax) {
    int alea = (int) (random()*(borneMax-borneMin+1));
    return borneMin + alea;
  }

  int random(int n) {
    return random(0, n);
  }

  void algorithm() {
    int min = 1;
    int max = 6;
    println(min+" <= "+random(min, max)+" <= " +max);
    println(min+" <= "+random(max)+" <= "+max);
  }
}
```

Correction d'un bug présent dans `random(int, int)`

```java
class RandomPratique extends Program {

   int random(int borneMin, int borneMax) {
      int alea = (int) (random()*(borneMax-borneMin+1));
      return borneMin + alea;
   }

   int
      r
   }

   voi
      i
      i
      p
      p
   }
}
```

```
--------------------------------------
Summary of RandomPratique Test Results

Your tests: 100% (2/2)
✅ testCopieSans
✅ testRandomN

Professor tests: 0% (0/0)

Total: 100% (2/2 yours | 0/0 professor)
```

VICTOIRE ! :)

+max);

Correction d'un bug présent dans la fonction `random`

ijava et les tests automatisés

- En fonction de la commande utilisée, ijava réagit différemment
  - `execute` : lance la fonction `void algorithm()`
  - `test` : recherche toutes les fonctions préfixées par `test` dans votre programme et les exécutent automatiquement, ainsi que les tests cachés des enseignant·e·s, puis génère une synthèse de l'ensemble des tests réalisés
- **Pensez à développer de manière incrémentale : écrire un test, puis la fonction correspondante avec un retour par défaut (test au rouge), puis écrivez le code fonctionnel jusqu'à avoir un test au vert**

Fonctions d'assertion

- Pour l'instant, nous n'utiliserons qu'une unique fonction d'assertion :
  - `void assertEquals(<valeurAttendue>, <valeurRetournéeParLaFonction>)`
  - Comme son nom l'indique cette fonction vérifie que les deux valeurs sont égales
  - **On place la valeur attendue en premier paramètre et l'appel à la fonction testée en second paramètre**
  - `assertEquals` est une fonction surchargée disponible pour tous les types que nous manipulons :)

Fonctions de tests

- Pour tester une fonction, on écrit **une fonction dont le nom commence par `test` et contenant une (ou plusieurs) assertion(s)**
- Ces vérifications sont réalisées à l'aide d'appel à la fonction testée avec des paramètres spécifiques pour lesquels nous connaissons les résultats
- On vérifie ensuite grâce à `assertEquals` que la valeur calculée par la fonction correspond bien à celle que l'on souhaitait

```
class CopieSans extends Program {

  String copieSans(String phrase, char symbole) { … }

  void  testCopieSans() {
    final String HELLO = "Hello";
    final String VIDE  = "";
    assertEquals("Hell",  copieSans(HELLO, 'o'));
    assertEquals("ello",  copieSans(HELLO, 'H'));
    assertEquals("Hello", copieSans(HELLO, 'a'));
    assertEquals("",      copieSans(VIDE,  'a'));
  }

  void _algorithm() { … }
}
```

Autre exemple : fonction de test pour copieSans

Synthèse

- **Une fonction est définie par**
  - une **signature** précisant son **nom**, les éventuels **paramètres** attendus et le **type du résultat** produit
  - un **corps** contenant les instructions réalisant le traitement
- La **surcharge** permet de définir plusieurs fonctions ayant le même nom, pour peu que leurs signatures diffèrent (plus précisément la liste des paramètres)

- **Les variables ont une portée définie par le bloc où elles sont déclarées**.
- Une variable globale se déclare en dehors de toute fonction : **à utiliser avec modération!**
- **Une assertion s'assure qu'une propriété est vérifiée** et enregistre l'erreur si ce n'est pas le cas
- **Une fonction de test doit avoir un nom préfixé par** `test` **et contenir au moins une assertion**