

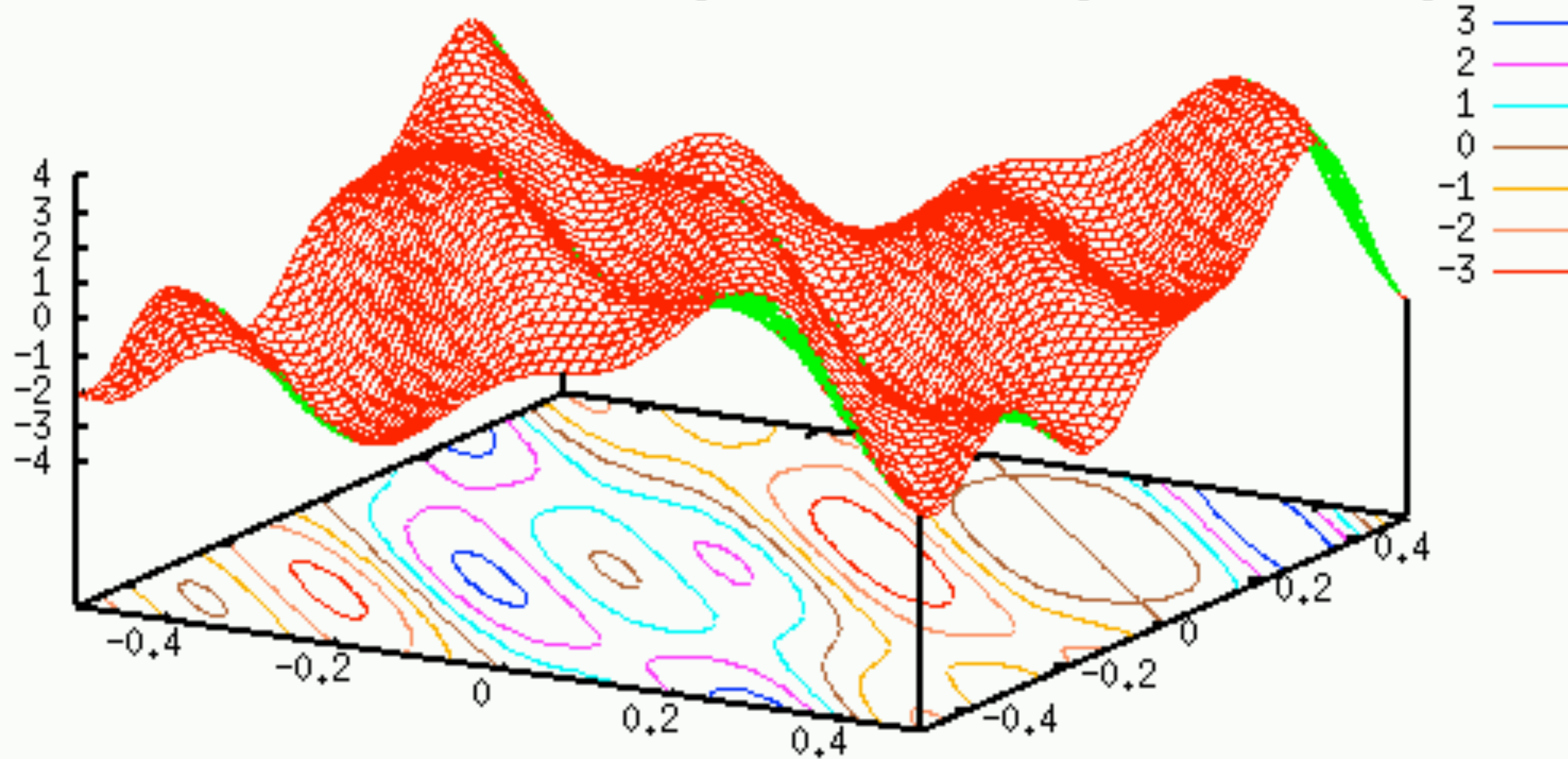
Algorithmique & Programmation

La notion de fonction

yann.secq@univ-lille.fr

ABIDI Sofiene, ALMEIDA COCO Amadeu, BONEVA Iovka, CASTILLON Antoine,
DELECROIX Fabien, LEPRETRE Éric, SANTANA MAIA Deise, SECQ Yann

$$f(z) = \cos(6.28*(3*x+2*y)) + \cos(6.28*(2*x+3*y)) - 2*\sin(6.28*(x+y))$$



SOFT DRINK / NEWSPAPER

A little rest is offered to reaching your destination



販売中



販売中

営業中

PASMO



100円

100円

100円

100円

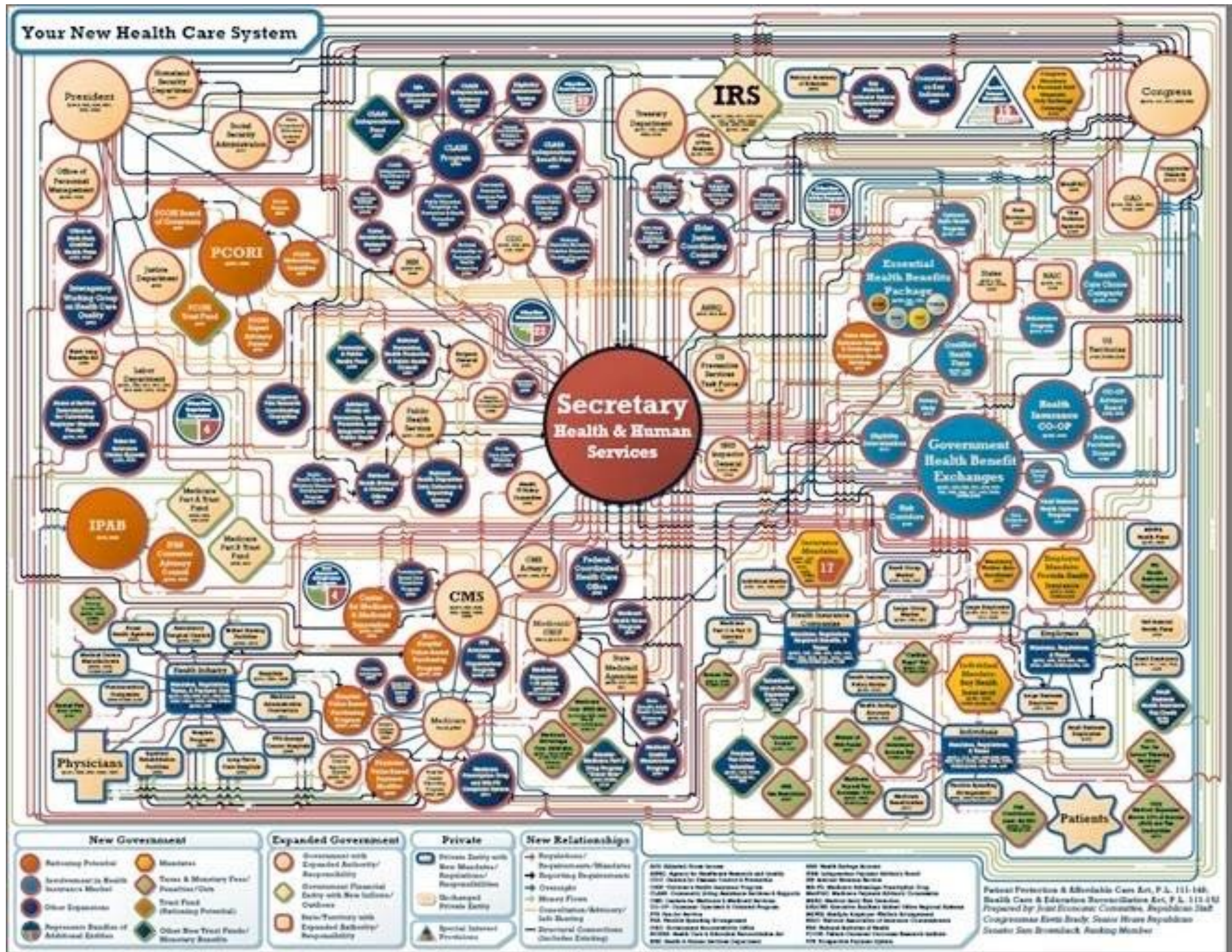
100円

News Papers

La notion de fonction

- Algorithme = [donnée(s) en entrée] + instructions + [donnée(s) en sortie]
- Fonction = algorithme réutilisable
- La notion de fonction permet de décomposer un algorithme complexe en un ensemble de sous-systèmes

Maîtriser la complexité = diviser pour régner !



Essence de la programmation ?

- Décomposition de systèmes complexes
- Identification de traitements récurrents
- Découpage d'un système en sous-systèmes
- Factorisation de code redondant
- Algorithme complexe = fugue d'appels de fonctions



Complexité (et intérêt !) de la programmation

- Comment structurer des programmes ?
- Comment faciliter leur création ?
- Comment améliorer leur qualité ?
- Comment réutiliser des algorithmes ?

Un exemple simple

```
class CorrigerTexte extends Program {  
  
    // définition de la fonction efface ...  
  
    void algorithm() {  
        String avant = "La disparition";  
        String apres = copieSans(avant, 'i');  
        println(apres); // La dspariton  
    }  
}
```

Un exemple simple

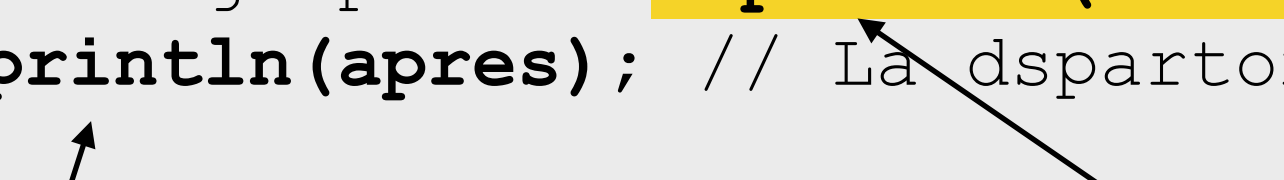
```
class CorrigerTexte extends Program {  
  
    // définition de la fonction efface ...  
  
    void algorithm() {  
        String avant = "La disparition »;  
        String apres = copieSans(avant, 'i');  
        println(apres); // La dsparton }  
    }
```

*Appel d'une fonction nommée
println avec une valeur
(une chaîne de caractères)*

*Appel d'une fonction
nommée copieSans
avec deux valeurs (une
chaîne et un caractère)*

Un exemple simple

```
class CorrigerTexte extends Program {  
  
    // définition de la fonction efface ...  
  
    void algorithm() {  
        String avant = "La disparition »;  
        String apres = copieSans(avant, 'i');  
        println(apres); // La dsparton }  
    }
```



**Fonction
prédéfinie**

**Fonction
à définir !**

Analyse de l'appel de fonction

```
String avant = "La disparition";  
String apres = copieSans(avant, 'i');
```

Type du
résultat
produit

Nom de la
fonction

Données nécessaires pour réaliser
le traitement (ou paramètres)

(avant, 'i')

Une chaîne de
caractères
String

Un caractère
char

String copieSans(String phrase, char lettre)

Un exemple simple

```
class CorrigerTexte extends Program {  
  
    // Signature de la fonction  
    String copieSans(String phrase, char lettre) {  
        // corps de la fonction  
    }  
  
    void algorithm() {  
        String avant = "La disparition »;  
        String apres = copieSans(avant, 'i');  
        println(apres); // La dsparton  
    }  
}
```

```
class CorrigerTexte extends Program {  
  
    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 avant = "La disparition »;  
        String apres = copieSans(avant, 'i');  
        println(apres); // La dsparton  
    }  
}
```

*Définition de
la fonction
copieSans*

Appel de la fonction copieSans

Une fonction peut appeler une autre fonction !

```
class CorrigerTexte extends Program {  
  
    String copieSans (String phrase, char lettre) {...}  
  
    String copieSansVoyelles (String phrase) {  
        String resultat = copieSans (phrase, 'a');  
        resultat = copieSans (resultat, 'e');  
        resultat = copieSans (resultat, 'i');  
        resultat = copieSans (resultat, 'o');  
        resultat = copieSans (resultat, 'u');  
        return resultat;  
    }  
  
    void algorithm() {  
        String avant = "La disparition";  
        String apres = copieSansVoyelles (avant);  
        println (apres); // L dsprtn  
    }  
}
```

La notion de fonction

- **Une fonction est un algorithme réutilisable**
- Une fonction nécessite des **informations en entrée** et produit **un résultat en sortie**
- Une fonction peut utiliser une autre fonction (ou elle même, mais on verra cela plus tard ...)
- Instructions que l'on connaît = fonctions prédéfinies !

Notion de portée

- Les variables ont une portée bien définie
- Une variable n'existe que dans le bloc où elle est déclarée
- Avant, la vie était simple ... (sauf `for` !)
- Maintenant, soyez attentifs à cette notion de portée avec les fonctions !


```

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

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

```

Et si resultat était nommée texte ?

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
algorithmme

```
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'));  
    }  
}
```

Synthèse

- **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 !**
- Importance de **la notion de portée des variables** et paramètres

Synthèse : signature et appel de fonction

```
String avant = "La disparition";  
String apres = copieSans(avant, 'i');
```

Type du
résultat
produit

Nom de la
fonction

Données nécessaires pour réaliser
le traitement (ou paramètres)

(avant, 'i')

Une chaîne de
caractères
String

Un caractère
char

```
String copieSans(String phrase, char lettre)
```

Nombre d'occurrences

```
class NbOccurrences extends Program {  
  
    void algorithm() {  
        String phrase = "La disparition";  
        char lettre    = 'i';  
        int nbOccurrences = 0;  
        for (int idx=0; idx < length(phrase); idx=idx+1) {  
            if (charAt(phrase,idx) == lettre) {  
                nbOccurrences = nbOccurrences + 1;  
            }  
        }  
        println("Il y a "+nbOccurrences+" fois la lettre "  
                +lettre+" dans \""+phrase+"\"");  
    }  
}
```

Déplacement du traitement dans une fonction !

Nombre d'occurrences

```
class NbOccurrences extends Program {  
  
    void algorithm() {  
        String phrase = "La disparition";  
        char lettre = 'i';  
        int nbOccurrences = 0;  
        for (int idx=0; idx < length(phrase); idx=idx+1) {  
            if (charAt(phrase,idx) == lettre) {  
                nbOccurrences = nbOccurrences + 1;  
            }  
        }  
        println("Il y a "+nbOccurrences+" fois la lettre "  
                +lettre+" dans \""+phrase+"\"");  
    }  
}
```

Déplacement du traitement dans une fonction !

Nombre d'occurrences

```
class NbOccurrences extends Program {  
    int nombreOccurrences(String mot, char lettre) {  
        int nbOccurrences = 0;  
        for (int idx = 0; idx < length(mot); idx=idx+1) {  
            if (charAt(mot, idx) == lettre) {  
                nbOccurrences = nbOccurrences + 1;  
            }  
        }  
        return nbOccurrences;  
    }  
    void algorithm() {  
        String phrase = "La disparition";  
        char symbole = 'i';  
        println("Il y a "+  
                nombreOccurrences(phrase, symbole) +  
                "fois la lettre "+symbole+" dans \""+phrase+"\"");  
    }  
}
```

Un random plus pratique

```
class RandomPratique extends Program {  
  
    void algorithm() {  
        int min = 1;  
        int max = 6;  
        int alea = min + (int) (random() * (max-min+1));  
        println(min+" <= "+alea+" <= "+max);  
    }  
  
}
```

Un random plus pratique

```
class RandomPratique extends Program {  
  
    int random(int borneMin, int borneMax) {  
        int alea = borneMin +  
                (int) (random() * (borneMax-borneMin+1)) ;  
        return alea;  
    }  
    void algorithm() {  
        int min = 1;  
        int max = 6;  
        println(min+" <= "+random(min, max)+" <= "+max) ;  
    }  
}
```

Déplacement du traitement dans une fonction !

Le jeu de Nim

- Jeu à information complète à 2 joueurs
- N allumettes, chacun en tire entre 1 et 3 par tour, celui prenant la dernière perd
- Comment décomposer la description de ce jeu pour faciliter sa programmation ?

Le jeu de Nim : analyse

- Jeu à information complète à 2 joueurs
- n allumettes, chacun en tire entre 1 et 3 par tour, celui prenant la dernière perd

● Quelles données ?

- Allumettes ? Joueurs ?
- Quels usages ? Quels types ? Variation ou pas ?

● Quels traitements ?

- Notion de tour de jeu
- Notion de fin de gain d'une partie



Algorithmique & Programmation

La notion de fonction (2)

yann.secq@univ-lille.fr

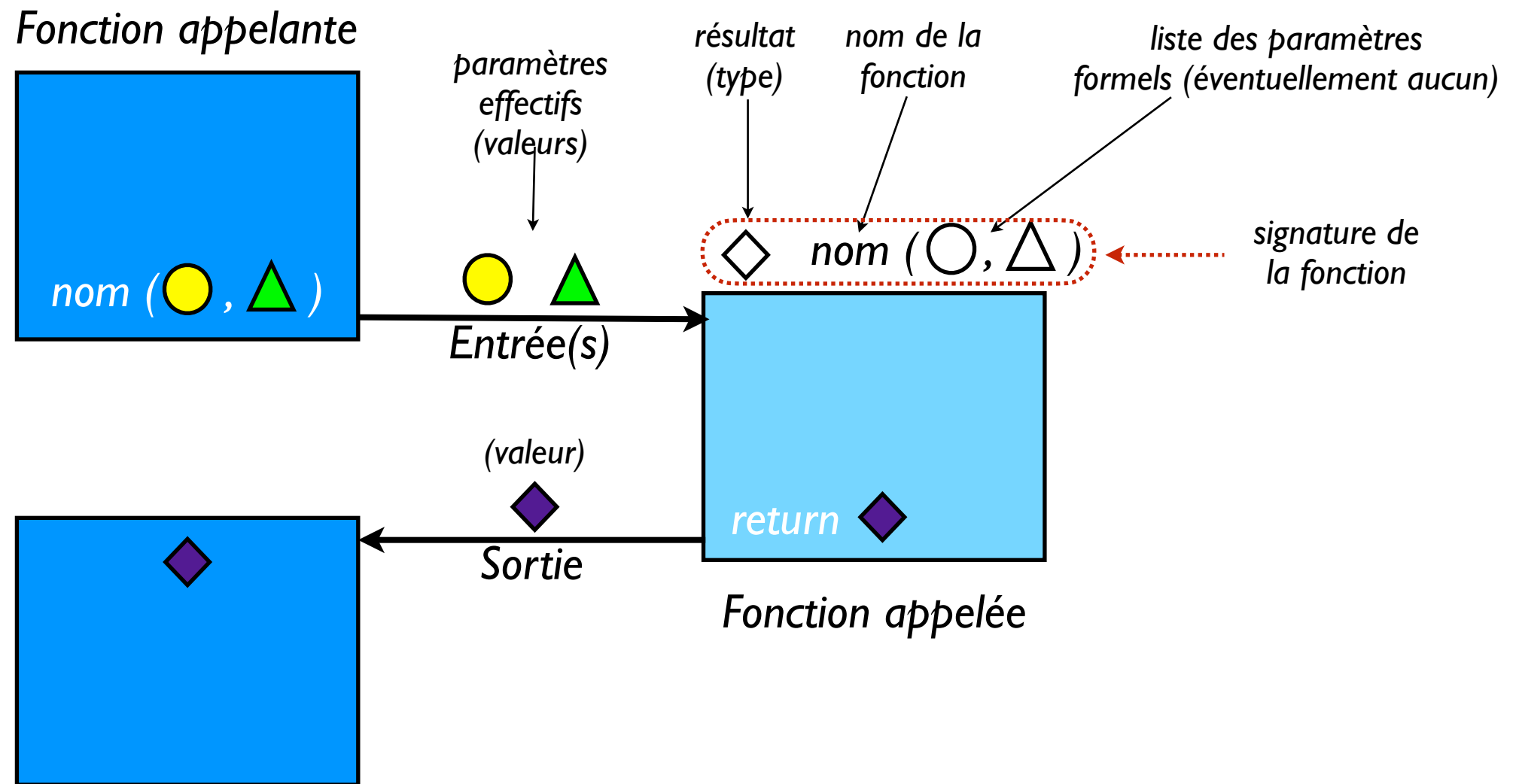
ABIDI Sofiene, ALMEIDA COCO Amadeu, BONEVA Iovka, CASTILLON Antoine,
DELECROIX Fabien, LEPRETRE Éric, SANTANA MAIA Deise, SECQ Yann

Notion de fonction

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

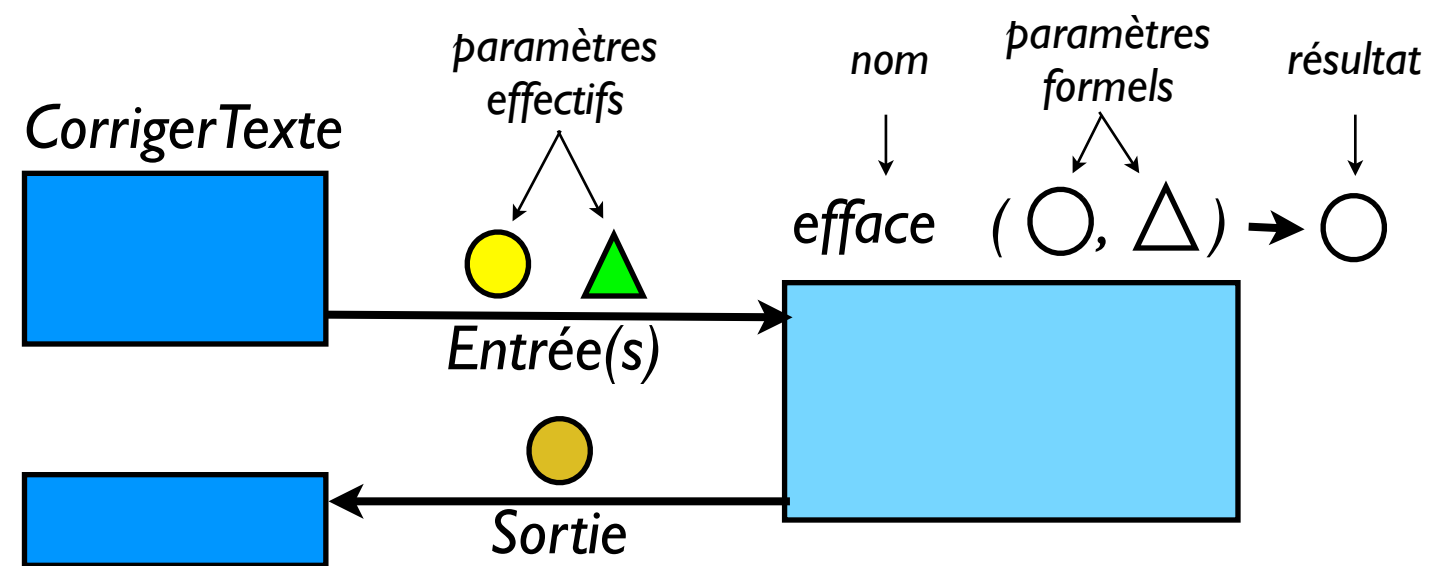
- **Une fonction est définie par sa signature**
 - le **nom** de la fonction (le plus pertinent possible !)
 - les **paramètres** (informations nécessaires pour réaliser le calcul)
 - le **type de son résultat**
- **ATTENTION : un seul return par fonction et toujours en dernière instruction !**

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
 - transmet les **valeurs** (paramètres *effectifs*) présentes dans l'appel en les associant aux paramètres (*formels*) 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 formels
msg et c

```
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'));
        ──────────── ● ────────────
    }
}
```

Paramètres effectifs
texte et 'e'

La valeur
contenue dans
la variable texte

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 !

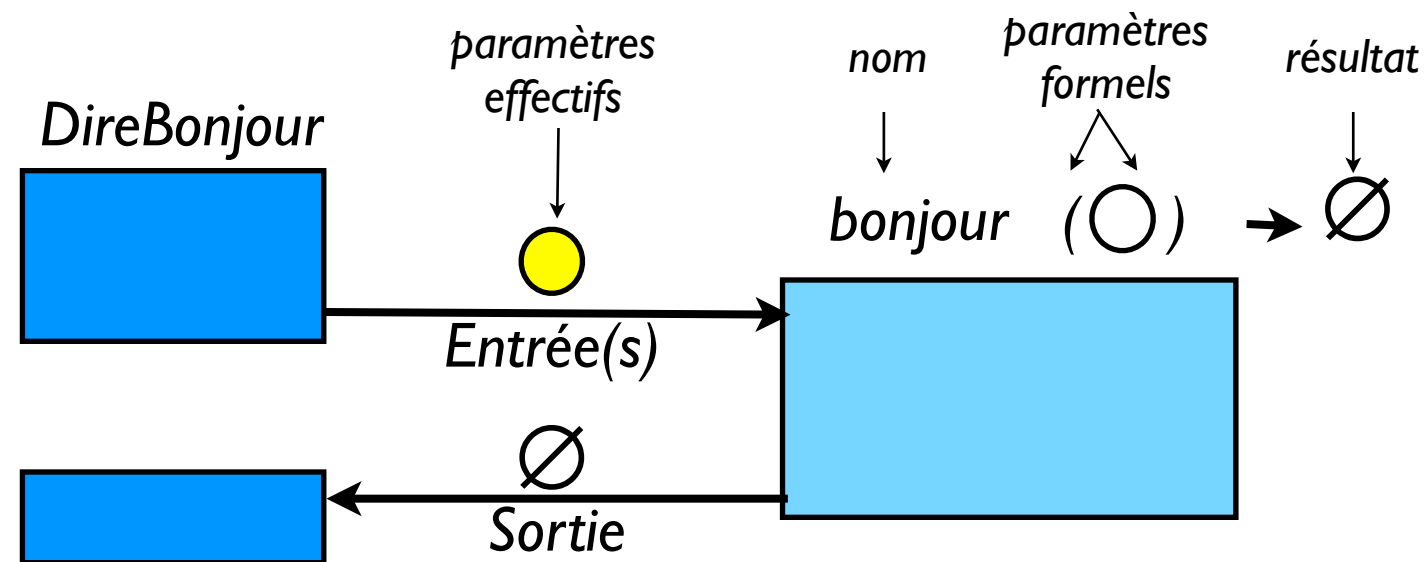
Passage par valeur

```
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 {
    Ø
    void bonjour(String nom) {
        println("Bonjour "+nom+" !");
    }

    void algorithm() {
        String patronyme;
        print("Veuillez entrer votre nom: ");
        patronyme = readString();
        Ø 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*

Attention : à ce renommage pour déclencher les tests !

Test de la fonction `random(int n)`

Affichage de l'exécution

Comment interpréter ces messages ?

```
• yannsecq@YANNs-MacBook-Pro fouillis % javac -cp ../ap.jar:. RandomSurcharge.java
• yannsecq@YANNs-MacBook-Pro fouillis % java -cp ../ap.jar:. RandomSurcharge
No function 'algorithm' found, running tests:
1> testCopieSans
2> testRandomN => AssertionError: [assertEquals] /= 6
1 test(s) verified on 2 tests (50% success).
○ yannsecq@YANNs-MacBook-Pro fouillis %
```

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)`

```

class RandomPratique extends Program {

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

```

```

● yannsecq@YANNs-MacBook-Pro fouillis % javac -cp ../ap.jar:. RandomSurcharge.java
● yannsecq@YANNs-MacBook-Pro fouillis % java -cp ../ap.jar:. RandomSurcharge
No function 'algorithm' found, running tests:
1> testCopieSans
2> testRandomN
2 test(s) verified on 2 tests (100% success).
○ yannsecq@YANNs-MacBook-Pro fouillis %

```

VICTOIRE ! :)

```

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

```

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

ijava et les tests automatisés

- Par défaut, *ijava* recherche la fonction `void algorithm()` et l'exécute si elle est présente
- Si `algorithm` n'est pas détectée, alors *ijava* recherche toutes les fonctions préfixées par `test` et les exécutent automatiquement, puis génère une synthèse des tests réalisés
- **N'oubliez pas d'ajouter un `_` à `algorithm` si vous souhaitez exécuter les tests** plutôt que le programme principal (et à l'enlever une fois que **tous les tests sont au vert** !)

Fonctions d'assertion

- Pour l'instant, nous n'utiliserons qu'une unique fonction d'assertion :
 - `void assertEquals(<valeur>, <valeur>)`
- 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 assertions**
- 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**
- Pour déclencher les tests, il faut renommer `algorithm` en `_algorithm`

