

IFT1015 Programmation 1

Abstraction procédurale

(Fonctions et procédures)

Marc Feeley

(avec ajouts de Aaron Courville et Pascal Vincent)

Abstraction procédurale

Abstraction procédurale

- On a souvent besoin d'exécuter les **mêmes opérations** dans des contextes différents
- **Exemple 1** : jeu vidéo qui doit afficher un même personnage dans le jeu et l'écran de configuration
- **Exemple 2** : logiciel d'une compagnie d'assurance qui doit imprimer des chèques de paye et des chèques d'indemnisation
- **Exemple 3** : logiciel de conception assistée par ordinateur qui doit calculer le périmètre d'un triangle connaissant les coordonnées 2D de ses 3 coins : $(x1, y1)$, $(x2, y2)$ et $(x3, y3)$

Abstraction procédurale

- Une approche serait de **dupliquer le code** qui effectue le traitement désiré à plusieurs endroits dans le programme :

```
// somme des racines carrées de 4 et 9

var n1 = 4; // calcul de la racine carrée de 4
var a1 = n1;
while (a1 > (a1+n1/a1)/2) {
    a1 = (a1+n1/a1)/2;
}

var n2 = 9; // calcul de la racine carrée de 9
var a2 = n2;
while (a2 > (a2+n2/a2)/2) {
    a2 = (a2+n2/a2)/2;
}

print(a1+a2); // additionner les racines
```


Abstraction procédurale

- Cela donne du **code difficile à comprendre et maintenir** car le programmeur ne s'exprime pas avec des termes qui sont proches des concepts propres à l'application
- Il y a **trop de détails** qui viennent obscurcir la logique du programme
- Comparer avec le programme suivant qui utilise directement la fonction de racine carrée prédéfinie :

```
// somme des racines carrées de 4 et 9  
print(Math.sqrt(4) + Math.sqrt(9)) ;
```


Abstraction procédurale

- Que faire s'il n'y a pas une fonction prédéfinie dans le langage pour le traitement désiré?
- **L'abstraction procédurale** permet au programmeur de définir ses propres opérations, à un niveau d'abstraction qui est plus proche des besoins de l'application
- Il y a plusieurs termes qui sont utilisés pour désigner des abstractions procédurales (**fonction, procédure, méthode, routine, sous-routine**)
- **Un aspect critique d'une bonne programmation est le choix des bonnes abstractions procédurales**

Abstraction procédurale

- En JS, l'abstraction procédurale peut se faire avec l'expression «**function**» qui définit une **fonction**
- Syntaxe : **function** (*<id.>*, ...) { *<énoncés>* }
- Les *<énoncés>* sont le **corps de la fonction**
- L'ensemble d'identificateurs entre parenthèses sont les **paramètres formels** de la fonction
- La **valeur** de l'expression «**function**» c'est une fonction (parfois aussi appelée «fermeture»)
- Les fonctions sont donc des valeurs, comme les nombres, textes et booléens

Abstraction procédurale

- Très souvent on combine l'expression «**function**» avec une **déclaration** de variable pour associer un nom à la fonction :

var *<identificateur>* = **function** (...) { *<énoncés>* } ;

- Par la suite on peut **appeler** la fonction avec la syntaxe habituelle d'appel de fonction en utilisant la variable liée à la fonction :

<identificateur> (*<expression>* , ...)

L'ensemble d'expressions entre parenthèses sont les **paramètres actuels** de la fonction

Abstraction procédurale

- Lorsqu'une fonction est appelée, les paramètres **actuels** sont évalués, des variables pour les paramètres **formels** sont créés et initialisés à la valeur des paramètres actuels correspondants, et finalement le **corps** de la fonction est exécuté

```
// affiche des informations sur deux cours
```

```
var rapport = function (titre, nbEtudiants) {  
  print("*** " + titre + " ***");  
  print(nbEtudiants + " étudiants");  
};
```

corps

paramètres
formels

appels de fonction

paramètres
actuels

```
rapport ("IFT1015", 125);  
rapport ("IFT2035", 51);
```


Abstraction procédurale

- Les paramètres formels sont **locaux** au corps de la fonction et n'ont aucun lien avec des variables de même nom à l'extérieur de la fonction
- Un identificateur désigne la déclaration (de variable ou de paramètre formel) qui est **la plus proche** en terme d'imbrication du code

```
// test de la portée des variables  
var x = 1;  
var test = function (x) { print(x); };  
test(2); // imprime 2  
print(x); // imprime 1
```


Abstraction procédurale

- Autres exemples :

```
// test de la portée des variables
```

```
var x = 1;
```

```
var test = function (n) { var x = n*3; print(x); };
```

```
test(2); // imprime 6
```

```
print(x); // imprime 1
```

```
// test de la portée des variables
```

```
var s = 0;
```

```
var x = 0;
```

```
var test = function (x,y) { x *= y; s += x; };
```

```
test(2, 3);
```

```
print("s=" + s + " x=" + x); // imprime s=6 x=0
```


Syntaxe abrégée

- Il existe une syntaxe abrégée pour déclarer des fonctions qui sont liées à des variables :

function *<identificateur>* (...) { *<énoncés>* }

est équivalent à :

var *<identificateur>* = **function** (...) { *<énoncés>* } ;

- Pour des raisons pédagogiques on utilisera la syntaxe plus longue qui permet de mieux comprendre certains concepts de programmation
- Dans beaucoup d'autres langages de programmation il existe seulement la forme abrégée

Syntaxe abrégée

- Ces deux déclarations sont donc équivalentes :

```
var rapport = function (titre, nbEtudiants) {  
    print("*** " + titre + " ***");  
    print(nbEtudiants + " étudiants");  
};
```

```
function rapport(titre, nbEtudiants) {  
    print("*** " + titre + " ***");  
    print(nbEtudiants + " étudiants");  
}
```



pas de ;

Fonctions et procédures

Fonctions

- Les **fonctions** sont des abstractions procédurales qui retournent une **valeur** (contrairement aux **procédures** qui ne retournent pas de valeur)
- C'est à l'aide de l'énoncé **return** à l'intérieur du corps d'une fonction, qu'on indique la valeur que la fonction retourne
- Syntaxe : **return** *<expression>* ;

```
// fonction de mise au carré  
  
var carre = function (x) {  
    return x*x;  
};  
  
print(carre(2) + carre(3)); // imprime 13
```


Énoncé return

- L'énoncé **return** est souvent le dernier énoncé dans le corps d'une fonction :

```
// calcul de la moyenne de deux nombres
```

```
var moyenne = function (n1, n2) {  
    return (n1+n2)/2;  
};
```

```
//  
// calcul de xn
```

```
var puissance = function (x, n) {  
    var r = 1;  
    for (var i=0; i<n; i++) r *= x;  
    return r;  
};
```

```
print(moyenne(5, 3) + puissance(5, 3)); // imprime 129
```


Énoncé return

- On peut mettre l'énoncé **return** n'importe où, et il cause une terminaison immédiate du corps de la fonction :

```
// fonction de valeur absolue

var abs = function (x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
};

print(abs(-33)); // imprime 33
```


Énoncé return

- Autre exemple : plus petit diviseur d'un entier n

```
// Retourne le plus petit entier d (autre que 1)
// qui divise exactement l'entier n.

var plusPetitDiviseur = function (n) {

    for (var d=2; d<=Math.sqrt(n); d++) {
        if (n % d == 0) {
            return d; // on a trouvé le plus
                      // petit diviseur... quitter
        }
    }

    return n; // le plus petit diviseur est
              // nécessairement n
};

print(plusPetitDiviseur(15)); // imprime 3
print(plusPetitDiviseur(17)); // imprime 17
```


Procédures

- Une **procédure** est une abstraction procédurale qui ne retourne **pas de valeur**
- L'énoncé **return** peut s'utiliser **sans expression** dans une procédure pour terminer abruptement l'exécution du corps de la procédure
- Syntaxe : **return;**

// (Mauvais) exemple de return dans une procédure

```
var debug = function (message) {  
    if (message == "") return;  
    print(message);  
    alert(message);  
};
```

```
debug("tout va bien!");  
debug("");
```


Variables locales

- Les déclarations de variables à l'intérieur du corps d'une fonction sont **locales à la fonction** :

```
// somme des racines carrées de 4 et 9

var sqrt = function (n) {

    var a = n;    // approximation de la racine de n

    while (a > (a + n/a)/2) {
        a = (a + n/a)/2;    // améliorer l'approximation
    }

    return a;
};

print(sqrt(4) + sqrt(9)); // additionner les racines
```

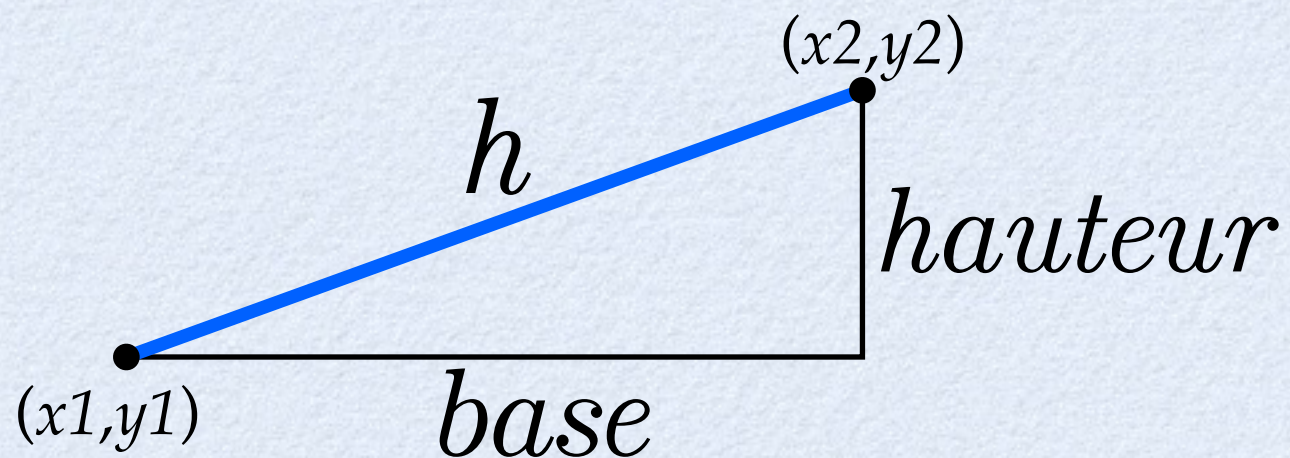
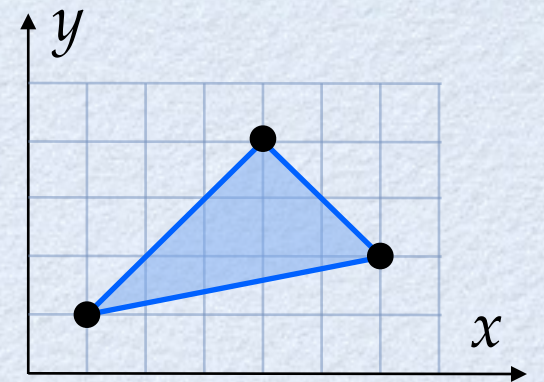

Décomposition fonctionnelle

Décomposition fonctionnelle

- Pour faciliter la compréhension du programme, il est bon de **décomposer le calcul en fonctions** qui font chacune une tâche spécifique
- Les fonctions devraient être **courtes**, de l'ordre de **10 à 20 lignes de code**, pour qu'il soit facile de comprendre leur fonctionnement en un clin d'oeil
- Une fonction (**principale**) peut faire appel à d'autres fonctions (**auxiliaires**) dans sa définition, et c'est normalement nécessaire pour décomposer en morceaux de taille raisonnable

Décomposition fonctionnelle

- Exemple : calcul du périmètre d'un triangle dont les coins sont aux coordonnées $(x1,y1)$, $(x2,y2)$ et $(x3,y3)$
- Il faut pouvoir calculer la distance entre 2 points $(x1,y1)$ et $(x2,y2)$
- Le calcul de distance entre 2 points peut se faire par le calcul de l'hypoténuse d'un triangle rectangle :



Théorème de Pythagore :

$$h = \sqrt{base^2 + hauteur^2}$$

Décomposition fonctionnelle

- Il faut donc des fonctions pour calculer :
 - le **périmètre** d'un triangle (fonction principale)
 - la **distance** entre 2 points
 - l'**hypoténuse** d'un triangle rectangle
 - le **carré** d'un nombre

Décomposition fonctionnelle

```
// Calcul du périmètre d'un triangle avec coins aux  
// coordonnées (x1,y1), (x2,y2), (x3,y3)
```

```
var perimetre = function (x1, y1, x2, y2, x3, y3) {  
    return distance(x1, y1, x2, y2) +  
           distance(x2, y2, x3, y3) +  
           distance(x3, y3, x1, y1);  
};
```

```
var distance = function (x1, y1, x2, y2) {  
    return hypotenuse(x2-x1, y2-y1);  
};
```

```
var hypotenuse = function (base, hauteur) {  
    return Math.sqrt(carre(base) + carre(hauteur));  
};
```

```
var carre = function (x) {  
    return x*x;  
};
```

```
print(perimetre(0,0, 4,0, 4,3)); // imprime 12
```


Tests unitaires

Tests unitaires

- Il est essentiel de **vérifier le comportement correct** des fonctions définies
- C'est **mieux de le faire tôt** dans le développement (aussitôt que la fonction est codée) pour éviter d'accumuler beaucoup de code non vérifié
- Cela peut se faire **interactivement** à la console (si on a un environnement de développement interactif comme codeBoot)
- Cependant ce n'est pas idéal, car lorsqu'on fait des modifications à notre code il faut à nouveau **refaire les vérifications** que le code modifié est correct

Tests unitaires

- Les **tests unitaires** sont des vérifications de comportement qui sont **incluses à même le code** (préféablement dans une fonction de test)
- Les tests seront **exécutés automatiquement** à chaque exécution du code (si les tests sont activés)
- En principe, **chaque fonction** définie devrait avoir ses propres tests unitaires, à moins que la fonction soit tellement simple que des tests soient inutiles
- Certaines fonctions complexes peuvent avoir des centaines de tests unitaires, particulièrement s'il y a plusieurs **cas spéciaux** à vérifier

Tests unitaires

- Un test unitaire d'une fonction est simplement une vérification qu'un appel de fonction spécifique donne le **résultat attendu**
- Par exemple, que **carre(-3) = 9**
- On pourrait donc faire, avec codeBoot :

```
var carre = function (x) {  
    return x*x;  
};  
  
var testCarre = function () {  
    if (carre(-3) != 9) alert("bogue : carre(-3) != 9");  
    if (carre(0) != 0) alert("bogue : carre(0) != 0");  
    // etc...  
};  
  
testCarre(); // mettre en commentaire pour désactiver
```


La fonction `assert`

- Cependant, cela demande beaucoup de codage pour chaque test, ce qui a comme effet de décourager le programmeur à écrire des tests
- La fonction **`assert`**, qui est prédéfinie dans codeBoot et d'autres environnements de développement, simplifie l'écriture de tests :

```
var carre = function (x) {  
    return x*x;  
};  
  
var testCarre = function () { // tests unitaires  
    assert( carre(-3) == 9 );  
    assert( carre(0)  == 0 );  
    // etc...  
};  
  
testCarre(); // mettre en commentaire pour désactiver
```


La fonction `assert`

- Le paramètre de **`assert`** est une valeur booléenne d'une condition qui **doit être vraie**
- Lorsque le paramètre de **`assert`** n'est pas vrai, l'exécution sera suspendue (comme avec un appel à **`pause`**) permettant ainsi d'identifier la situation où le problème survient
- On pourrait définir **`assert`** approximativement comme ceci :

```
var assert = function (test) {  
    if (test == false) {  
        pause();  
    }  
};
```


Exemple complet

```
// Calcul du périmètre d'un triangle avec coins aux
// coordonnées (x1,y1), (x2,y2), (x3,y3)

var perimetre = function (x1, y1, x2, y2, x3, y3) {...};
var distance = function (x1, y1, x2, y2) {...};
var hypotenuse = function (base, hauteur) {...};
var carre = function (x) {...};

var testPerimetre = function () { // tests unitaires

    assert( carre(-3) == 9 );
    assert( carre(0) == 0 );

    assert( hypotenuse(3,4) == 5 );
    assert( hypotenuse(5,12) == 13 );

    assert( distance(8,18, 16,12) == 10 );

    assert( perimetre(0,0, 4,0, 4,3) == 12 );
};

testPerimetre(); // mettre en commentaire pour désactiver
```


Gestion des tests unitaires

- On a avantage à concevoir les tests **avant de coder** la fonction car les tests qui échouent guident le programmeur dans son codage
- Les tests aident aussi à **documenter** le code
- Si on découvre un bogue avec le code existant qui n'a pas été détecté par les tests (i.e. les tests sont incomplets), il est important d'**ajouter un test pour cette situation avant de régler le bogue**
- Lorsqu'une modification au code fait qu'un test qui réussissait avant échoue (une dégradation du code), on dit que c'est une **régression** et il faut régler ce bogue prioritairement

Conception de fonctions

Conception de fonctions

- Il y a des **questions de conception de base** à se poser lors de la conception d'une fonction :
- **Q1** : Quelle est la **tâche** de la fonction?
- **Q2** : Quels sont les **paramètres** de la fonction (les données sur lesquelles la fonction se base pour calculer son résultat), et quels sont leurs **types** (nombre quelconque, entier, texte, booléen, ...)?
- **Q3** : Quel est le **type du résultat**?
- **Q4** : Quel est le meilleur **nom pour la fonction**?
- **Q5** : Quels sont des **exemples d'utilisation**?

Exemple : encodage de nombres

- Exemple : conversion d'un nombre entier à son encodage binaire
- Q1 : La fonction fait l'**encodage binaire**
- Q2 : L'unique paramètre est le nombre à convertir, qui est un **entier ≥ 0**
- Q3 : Le résultat est un **texte contenant des 0 et 1**
- Q4 : Le nom de la fonction est **encodageBinaire**
- Q5 : **encodageBinaire (13) = "1101", ...**
- Ces informations ont avantage à se trouver dans le code possiblement comme **commentaires et tests**

Exemple : encodage de nombres

- Dans la technique de **développement piloté par les tests** (“test driven development” ou TDD), les tests unitaires sont écrits en premier
- Les tests aident à préciser la spécification et peuvent la remplacer dans des situations simples
- Par exemple, on peut débiter avec ce code :

```
var testEncodageBinaire = function () { // tests unitaires
    assert( encodageBinaire(13) == "1101" ); // ces tests
    assert( encodageBinaire(0) == "0" );      // échouent!
};

testEncodageBinaire();
```


Exemple : encodage de nombres

- On peut ensuite écrire un **squelette** de la fonction recherchée qui retourne une **valeur incorrecte**
- Et on vérifie que les tests unitaires échouent

```
var encodageBinaire = function (n) {  
  
    // Cette fonction prend un paramètre n qui doit être  
    // un entier >= 0, et retourne un texte composé de 0  
    // et 1 qui est l'encodage binaire de n.  
  
    return ""; // TODO: compléter cette fonction  
};  
  
var testEncodageBinaire = function () { // tests unitaires  
    assert( encodageBinaire(13) == "1101" ); // ces tests  
    assert( encodageBinaire(0) == "0" );      // échouent!  
};  
  
testEncodageBinaire();
```


Exemple : encodage de nombres

- Il faut ensuite **raffiner le code** en étapes successives pour qu'il passe de plus en plus des tests unitaires, et éventuellement tous
- Les tests unitaires vont donc **guider les étapes de conception et codage**
- Il est donc important de choisir des tests unitaires pour des **cas usuels** mais aussi des **cas qui risquent de poser problème** (les cas "limite", les cas "spéciaux", etc)

Exemple : encodage de nombres

```
var encodageBinaire = function (n) {  
  
    // Cette fonction prend un paramètre n qui doit être  
    // un entier >= 0, et retourne un texte composé de 0  
    // et 1 qui est l'encodage binaire de n.  
  
    var e = ""; // pour accumuler l'encodage binaire  
  
    while (n > 0) { // tant qu'il reste des bits  
        e = (n % 2) + e; // accumuler un bit  
        n = Math.floor(n / 2); // passer aux autres bits  
    }  
  
    return e;  
};  
  
var testEncodageBinaire = function () { // tests unitaires  
    assert( encodageBinaire(13) == "1101" );  
    assert( encodageBinaire(0) == "0" ); // ce test échoue  
};  
  
testEncodageBinaire();
```


Exemple : encodage de nombres

```
var encodageBinaire = function (n) {  
  
    // Cette fonction prend un paramètre n qui doit être  
    // un entier >= 0, et retourne un texte composé de 0  
    // et 1 qui est l'encodage binaire de n.  
  
    var e = ""; // pour accumuler l'encodage binaire  
  
    do {  
        e = (n % 2) + e; // accumuler un bit  
        n = Math.floor(n / 2); // passer aux autres bits  
    } while (n > 0); // tant qu'il reste des bits  
  
    return e;  
};  
  
var testEncodageBinaire = function () { // tests unitaires  
    assert( encodageBinaire(13) == "1101" );  
    assert( encodageBinaire(0) == "0" );  
};  
  
testEncodageBinaire();
```


Exemple : plus grand commun diviseur

- Le PGCD de deux nombres entiers x et y c'est le plus grand nombre entier d qui divise exactement x et y (par exemple le PGCD de 18 et 30 c'est 6)
- Q1 : La fonction calcule le **PGCD de 2 entiers**
- Q2 : Les 2 paramètres sont les **2 entiers** $(x \text{ et } y) \geq 0$
- Q3 : Le résultat est un **entier**
- Q4 : Le nom de la fonction est **pgcd**
- Q5 : **pgcd (18 , 30) = 6, ...**

Exemple : plus grand commun diviseur

- On commence avec l'écriture des tests qui couvrent des cas usuels et spéciaux :

```
var testPgcd = function () { // tests unitaires
    assert( pgcd(18,30) == 6 ); // test échoue
    assert( pgcd(30,18) == 6 );
    assert( pgcd(8,15)  == 1 );
    assert( pgcd(9,1)   == 1 );
    assert( pgcd(0,8)   == 8 );
};

testPgcd();
```


Exemple : plus grand commun diviseur

- On code un squelette de la fonction :

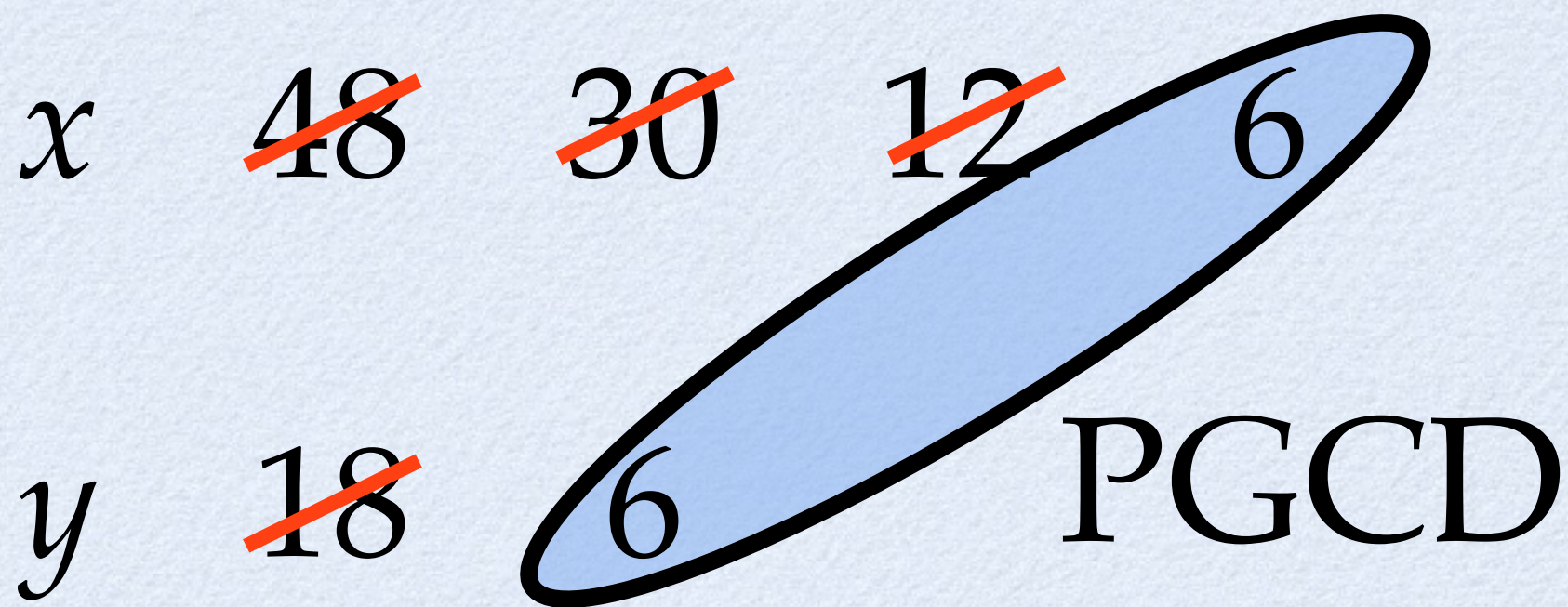
```
var pgcd = function (x, y) {  
    return 0; // TODO: compléter cette fonction  
};  
  
var testPgcd = function () { // tests unitaires  
    assert( pgcd(18,30) == 6 ); // test échoue  
    assert( pgcd(30,18) == 6 );  
    assert( pgcd(8,15)  == 1 );  
    assert( pgcd(9,1)   == 1 );  
    assert( pgcd(0,8)   == 8 );  
};  
  
testPgcd();
```


Exemple : plus grand commun diviseur

- Il faut trouver **comment calculer** le PGCD de x et y
- **Algorithme** : description abstraite de la procédure à suivre pour résoudre un problème
- Il existe plusieurs algorithmes pour le PGCD
- Un des plus simples est l'**algorithme d'Euclide**
- Par la suite l'algorithme sera **codé** dans un langage de programmation précis, ici en JS

Exemple : plus grand commun diviseur

- **Algorithme d'Euclide** : soustraire le plus petit de x et y de l'autre nombre et répéter jusqu'à ce qu'ils soient égaux, ce sera le PGCD de x et y




Exemple : plus grand commun diviseur

- On code l'algorithme et on le teste :

```
var pgcd = function (x, y) {  
    while (x != y) { // algorithme d'Euclide  
        if (x < y) {  
            y -= x;  
        } else {  
            x -= y;  
        }  
    }  
    return x;  
};  
  
var testPgcd = function () { // tests unitaires  
    assert( pgcd(18,30) == 6 );  
    assert( pgcd(30,18) == 6 );  
    assert( pgcd(8,15) == 1 );  
    assert( pgcd(9,1) == 1 );  
    assert( pgcd(0,8) == 8 ); // boucle infinie  
};  
  
testPgcd();
```


il y a un
bogue!



Exemple : plus grand commun diviseur

- On élimine le bogue en traitant les cas spéciaux :

```
var pgcd = function (x, y) {  
  if (x == 0) return y; // cas spéciaux  
  if (y == 0) return x;  
  while (x != y) { // algorithme d'Euclide  
    if (x < y) {  
      y -= x;  
    } else {  
      x -= y;  
    }  
  }  
  return x;  
};  
  
var testPgcd = function () { // tests unitaires  
  assert( pgcd(18,30) == 6 );  
  assert( pgcd(30,18) == 6 );  
  assert( pgcd(8,15) == 1 );  
  assert( pgcd(9,1) == 1 );  
  assert( pgcd(0,8) == 8 );  
};  
  
testPgcd();
```



maintenant
tous les tests
passent

Exemple : plus grand commun diviseur


- On améliore l'algorithme pour l'accélérer :

```
var pgcd = function (x, y) {  
    while (x != 0 && y != 0) { // algorithme d'Euclide  
        if (x < y) {  
            y = y % x;  
        } else {  
            x = x % y;  
        }  
    }  
    if (x == 0) return y; else return x;  
};
```

```
var testPgcd = function () { // tests unitaires  
    assert( pgcd(18,30) == 6 );  
    assert( pgcd(30,18) == 6 );  
    assert( pgcd(8,15)  == 1 );  
    assert( pgcd(9,1)   == 1 );  
    assert( pgcd(0,8)   == 8 );  
};
```

```
testPgcd();
```

les tests
passent
toujours, ce qui
nous rassure
que nos
changements
n'ont rien brisé




Exemple : plus grand commun diviseur

- On améliore encore l'algorithme pour l'accélérer :

```
var pgcd = function (x, y) {  
  var t;  
  if (x > y) { t = x; x = y; y = t; }  
  while (x != 0) { // algorithme d'Euclide  
    t = x;  
    x = y % x;  
    y = t;  
  }  
  return y;  
};  
  
var testPgcd = function () { // tests unitaires  
  assert( pgcd(18,30) == 6 );  
  assert( pgcd(30,18) == 6 );  
  assert( pgcd(8,15)  == 1 );  
  assert( pgcd(9,1)   == 1 );  
  assert( pgcd(0,8)   == 8 );  
};  
  
testPgcd();
```

les tests
passent
toujours, ce qui
nous rassure
que nos
changements
n'ont rien brisé



Exemple : plus grand commun diviseur

- L'exécution systématique des tests unitaires à **chaque changement du code** permet de garder le code dans un état de fonctionnement tout au long de son développement
- On commence par une version du code qui est correcte, mais possiblement pas très performante
- On y apporte des améliorations et extensions par la suite

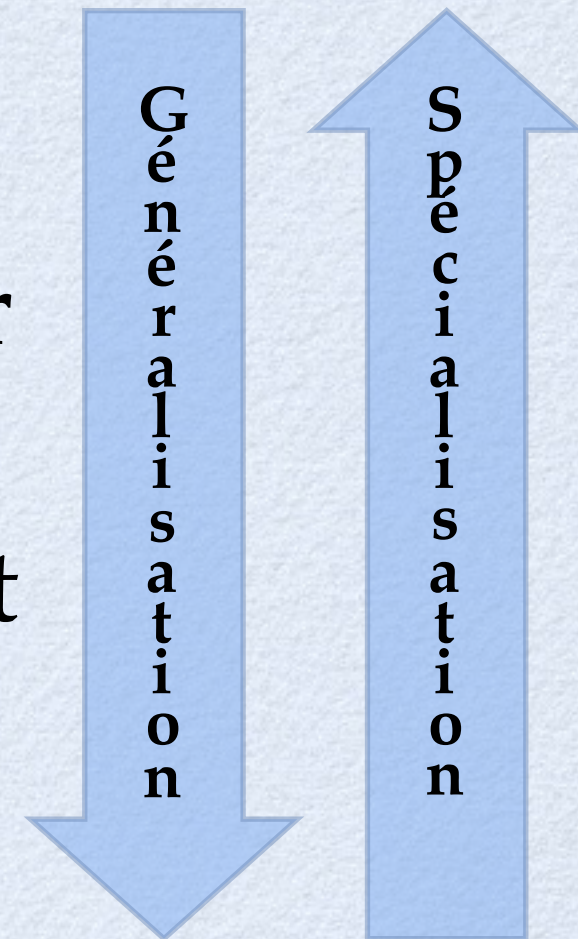
Spécialisation et généralisation

- Une fonction conçue pour un programme peut être **plus ou moins spécifique à ce programme**
- C'est le **degré de spécialisation** de la fonction
- L'inverse de la spécialisation c'est la **généralisation**
- Une fonction g est plus générale qu'une fonction f (et f est plus spécifique que g) si avec g on peut faire **tous les calculs possibles avec f et plus** (en d'autres termes g couvre plus de contextes d'utilisation que f)
- Exemple : fonction pour encoder en binaire et une fonction pour encoder en n'importe quelle base

Spécialisation et généralisation

```
var encodageBinaire = function (n) {...};  
var encodagePositionnel = function (n, base) {...};  
var encodagePositionnelAvecSigne = function (n, base, signe) {...};
```

- **encodageBinaire** permet d'encoder en base 2 un entier $n \geq 0$
- **encodagePositionnel** permet d'encoder en une base entre 2 et 10 un entier $n \geq 0$
- **encodagePositionnelAvecSigne** permet d'encoder en une base entre 2 et 10 un entier quelconque et optionnellement forcer un signe + ou - devant
- Évidemment, d'autres spécialisations et généralisations sont possibles



Spécialisation et généralisation

- L'attrait de concevoir des fonctions plus générales c'est qu'on peut plus facilement les **réutiliser** dans d'autres contextes (que ce soit dans le programme même, dans une extension du programme, ou d'autres programmes)
- Une fonction spécialisée est normalement **plus facile à concevoir et plus performante**, mais il y a de fortes chances qu'il faudra la remplacer par une fonction plus générale lors de l'évolution du code
- Il est donc utile d'investir le temps nécessaire à la **conception d'une fonction générale dès le début** (en d'autres termes de prévoir les besoins futurs)


```

var encodagePositionnelAvecSigne = function (entier, base, signe) {

    var n = Math.abs(entier); // nombre non-négatif à encoder
    var e = "";              // pour accumuler l'encodage

    do {
        e = (n % base) + e; // accumuler un chiffre
        n = Math.floor(n / base); // passer aux autres chiffres
    } while (n > 0); // tant qu'il reste des chiffres

    if (entier < 0) { // ajouter le signe lorsque nécessaire
        return "-" + e;
    } else if (signe) {
        return "+" + e;
    } else {
        return e;
    }
};

```

```

var testEncodage = function () { // tests unitaires
    assert( encodagePositionnelAvecSigne(13,2,false) == "1101" );
    assert( encodagePositionnelAvecSigne(13,10,true) == "+13" );
    assert( encodagePositionnelAvecSigne(-16,8,false) == "-20" );
    assert( encodagePositionnelAvecSigne(0,10,true) == "+0" );
};

```

```

testEncodage();

```



```

var encodagePositionnelAvecSigne = function (entier, base, signe) {

    var e = encodagePositionnel(Math.abs(entier), base);

    if (entier < 0) {          // ajouter le signe lorsque nécessaire
        return "-" + e;
    } else if (signe) {
        return "+" + e;
    } else {
        return e;
    }
};

var encodagePositionnel = function (n, base) {

    var e = "";              // pour accumuler l'encodage

    do {
        e = (n % base) + e;   // accumuler un chiffre
        n = Math.floor(n / base); // passer aux autres chiffres
    } while (n > 0);          // tant qu'il reste des chiffres

    return e;
};

var testEncodage = function () {...}; // tests unitaires

testEncodage();

```


Spécialisation et généralisation

- La **généralisation** d'une fonction spécifique entraîne normalement l'**ajout de paramètres** ou l'élargissement du type des paramètres
- Les paramètres permettent d'ajuster le calcul effectué aux besoins du site d'appel
- Exemple : programme qui cherche à calculer $5 \times 5 \times 5$

```
var calcul = function () {  
    return 5*5*5;  
};  
  
print(calcul()); // imprime 125
```


Spécialisation et généralisation

- Généralisation par l'ajout d'un paramètre :

```
var cube = function (x) {  
    return x*x*x;  
};  
  
print(cube(5)); // imprime 125
```

- Autre généralisation par l'ajout d'un paramètre :

```
var expo = function (x, n) {  
    var r = 1;  
    for (var i=0; i<n; i++) r *= x;  
    return r;  
};  
  
print(expo(5, 3)); // imprime 125
```


Spécialisation et généralisation

- Généralisation par extension du type de **n** (de entier non négatif à réel non négatif) :

```
var expo = function (x, n) {  
    return Math.exp(n*Math.log(x)) ;  
};  
  
print(expo(5, 3)) ; // imprime 124.999999999999994
```


Spécialisation et généralisation

- Amélioration de la précision du calcul lorsque **n** est un entier :

```
var expo = function (x, n) {  
  if (n == Math.floor(n)) {  
    var r = 1;  
    for (var i=0; i<n; i++) r *= x;  
    return r;  
  } else {  
    return Math.exp(n*Math.log(x));  
  }  
};  
  
print(expo(5, 3)); // imprime 125
```