R1.01 : Initiation au dévelopement

Olivier ROUSSEL
olivier.roussel@univ-artois.fr

06/11/2023, version 4442

Le Programme National

Objectif

- L'objectif de cette ressource est l'initiation au développement. Cette ressource est nécessaire pour la réalisation d'un développement d'application et l'optimisation des applications informatiques.

Savoirs de référence étudiés

- Algorithmes fondamentaux (structures simples, recherche d'un élément, parcours, tri...)
- Algorithmes sur les structures de données (itératifs et/ou récursifs)
- Manipulation de listes, tableaux, collections dynamiques, statiques (accès direct ou séquentiels), piles, files, structures
- Types abstraits de données simples : première approche de l'encapsulation
- ...

Le Programme National (suite)

- ...
- Notions de modularité
- Premières notions de qualité (par ex. : nommage, assertions, documentation, sûreté de fonctionnement, jeu d'essais, performance...)
- Lecture/écriture de fichiers

Prolongements suggérés

- Introduction à la gestion de versions

Mots-clefs : Algorithmique, Structures de contrôle, Qualité de codage, Typage

Deux parties

- première moitié du semestre : les bases (O. Roussel)
- seconde moitié du semestre : notions plus avancées (A. Chmeiss)

Contenu de la première partie

- structures de contrôle
- variables, types, affectations
- tableaux, algorithmes sur les tableaux
- ...

Modalités de la première partie

Des chiffres :

- 8 semaines de cours,
- 6 heures par semaine de TD/TP
- 7 séances de cours,
- 2 DS en CM et de multiples contrôles surprise qui donneront une note de DS.

Les règles en CM, TD :

- `Vous devez utiliser votre cerveau !`
- Aucun appareil électronique (ordinateur, calculatrice, téléphone, montre connectée, ...)

Les règles pour les DS :

- Aucun appareil électronique (ordinateur, calculatrice, téléphone, ...)
- **Seul document autorisé : une feuille manuscrite, format A4, recto-verso, avec votre nom dessus**

TD/TP

- Les encadrés de cours dans les sujet de TD/TP donnent les informations essentielles (souvent vitales) que vous devez connaître impérativement.
- En TD, on travaillera uniquement sur papier, pour vous forcer à **réfléchir**
- En TP, on travaillera sur machine pour se confronter à la réalité de la machine (mais il faut **d'abord réfléchir** et ensuite seulement écrire le programme.
- On a très souvent besoin de dessiner des schémas pour comprendre ce que l'on fait.
- Un programmeur digne de ce nom sait ce qu'un programme fait en lisant le code.
- Vous ne devez pas jamais écrire de programme en faisant des modifications au hasard. Tout doit être réfléchi !

Avertissement !

- Certains transparents de ce cours contiennent un certain nombre de simplifications...
- ...donc un certain nombre de mensonges

Objectif

- Le but de ce cours est de vous faire maîtriser les bases de la programmation procédurale.
- Même si l'on utilise Java en TP, **ce n'est pas un cours de Java !**
- Les notions seront illustrées
  - en pseudo langage (à maîtriser)
  - en Java (à maîtriser)
  - parfois dans d'autres langages (pour information et mise en perspective)

Structures de contrôle

Programmation impérative

- Un programme est une séquence d'instructions.
- Une instruction est un ordre donné à la machine. Cet ordre permet de modifier l'état de la machine (affichage, lecture, modification de variables, pilotage de périphériques, ...)
- L'ordre d'exécution des instructions est donné par l'ordre d'écriture dans le programme.
- Globalement, un programme est une recette de cuisine. On manipule des ingrédients (les données) en effectuant des opérations (instructions) à réaliser dans un ordre spécifique.

Initialisation, traitement, finalisation

Un programme comporte toujours 3 phases (parfois implicitement)

- L'initialisation :
  On place la machine dans un état connu, qui nous permettra de faire le travail (sortir les ustensiles avant de cuisiner, vérifier qu'ils sont propre, etc.)
- Le traitement :
  Il consiste à effectuer des opérations successives sur les données pour produire le résultat attendu (suivre les instructions de la recette de cuisine)
- La finalisation :
  Il consiste à remettre la machine dans un état stable. En particulier, on libérera les ressources que l'on a utilisées (faire la vaisselle à la fin du repas).

Structure d'un programme Java

- Voici le squelette que vous utiliserez dans ce cours. Votre fichier s'appellera **NomDuProgramme**.java

```
import java.io.*;

class NomDuProgramme {
  // les variables "globales", les "fonctions"
  // et "procédures"
  ...
  void run() {
    // le code de votre programme
    ...
  }
  // lancer run() en évitant ``au mieux'' les objets
  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
}
```


Compilation et lancement d'un programme Java

- Le nom de votre programme (**NomDuProgramme**) commencera par une lettre, suivie d'autres lettres, chiffres ou caractère souligné.
- Ce programme Java est écrit dans un fichier texte **NomDuProgramme**.java
- Il doit être traduit en code compréhensible par le processeur. C'est la compilation, qui génère un ou des fichiers .class.

```
javac NomDuProgramme.java
```

- On peut alors exécuter le programme en tapant

```
java NomDuProgramme
```


L'instruction

- Une instruction est un ordre donné à l'ordinateur.
- Cette instruction va modifier l'état du programme.
- Exemples :
  - afficher une information,
  - modifier une variable,
  - changer le contenu d'un fichier,
  - ...
- En Java, toutes les instructions vont se terminer par un point-virgule.

La séquence d'instructions

- Une séquence d'instructions est une succession d'instructions qui doivent être exécutées les une après les autres, dans l'ordre dans lequel elle sont écrites dans le programme.
- Les instructions sont écrites à raison de une instruction par ligne.
- La séquence d'instructions est délimitée par une accolade ouvrante et une accolade fermante.
- Les instructions dans la séquence doivent être indentées de la même manière (même nombre d'espaces avant le premier caractère)
- Exemple :

```
{
  instruction1;
  instruction2;
  instruction3;
}
```


Les boucles

- Dans un programme, on doit souvent exécuter plusieurs fois certaines instructions.
- On utilise pour cela une boucle, c'est à dire une instruction qui indique comment répéter les instructions, et quelles instructions répétées.
- Il existe plusieurs types de boucles (for, foreach, while, do...while, ...)
- Chaque exécution des instructions de la boucle est appelée une *itération*
- En Java, les instructions sous le contrôle de la boucle seront placées dans un séquence d'instructions, donc entre accolades et indentées de la même manière.

La boucle pour (for)

- Quand on veut répeter des instructions un nombre de fois qui est connu juste avant le début de la boucle, on utiliser une boucle for
- Pour compter les itérations, on utilise un compteur de boucle, qui va prendre successivement les valeurs possibles dans un intervalle que l'on spécifie.
- Le nom du compteur de boucle doit commencer par une lettre (ou un souligné) et peut se poursuivre par des lettres, des chiffres ou des soulignés.
- On ne doit pas modifier ce compteur dans la boucle.
- Les instructions sont répétées pour chaque valeur possible du compteur.
- Avec une boucle for, le nombre d'itération est (normalement) connu à l'avance.

La boucle for (Java)

- En Java, pour répéter n fois une séquence, on compte habituellement de 0 à $n-1$ (bornes comprises) :

```
for (int nomCompteur = 0; nomCompteur < n; nomCompteur++) {
  instruction1;
  instruction2;
  ...
}
```

- Exemple :

```
// afficher 10 espaces
for (int i = 0 ;i < 10; i++) {
  System.out.print(" ");
}
```


La boucle for croissante (Java)

- En Java, pour compter de a à b (bornes comprises) de manière croissante :

```
for (int nomCompteur = a; nomCompteur <= b; nomCompteur++) {
  instruction1;
  instruction2;
  ...
}
```

- On peut utiliser la valeur du compteur dans la boucle. Le nom du compteur sera remplacé par sa valeur.
- Exemple :

```
System.out.println("Voici la liste des entiers de 1 à 10");
for (int i = 1; i <= 10; i++) {
  System.out.println(i);
}
```

- Si $b < a$, l'intervalle de valeurs est vide, et les instructions ne sont pas exécutées.

```
for (int i = 1; i <= 0; i++) {
  System.out.println(i); // jamais exécuté
}
```

La boucle for décroissante (Java)

- En Java, pour décompter de *a* à *b* (bornes comprises) de manière décroissante :

```
for (int  nomCompteur = a;   nomCompteur >=  b; nomCompteur--) {
  instruction1;
  instruction2;
  ...
}
```

- On peut utiliser la valeur du compteur dans la boucle. Le nom du compteur sera remplacé par sa valeur.
- Exemple :

```
System.out.println("Décompte de 5 à 0");
for (int i = 5; i >=0; i--) {
  System.out.println(i);
}
```

- Si $b > a$, l'intervalle de valeurs est vide, et les instructions ne sont pas exécutées.

```
for (int i = 5; i >= 6; i--) {
  System.out.println(i); // jamais exécuté
}
```


Pas d'une boucle

- En Java, pour compter de *a* à *b* (bornes comprises) de manière croissante en ajoutant *p* (le pas) au compteur à chaque itération :

```
for (int  nomCompteur = a;   nomCompteur <=  b; nomCompteur += p) {
  instructions;
}
```

- En Java, pour décompter de *a* à *b* (bornes comprises) de manière décroissante en retirant *p* (le pas) au compteur à chaque itération :

```
for (int  nomCompteur = a;   nomCompteur >=  b; nomCompteur -= p) {
  instructions;
}
```

- Exemple :

```
System.out.println("Nombres pairs de 2 à 10");
for (int i = 2; i <= 10; i += 2) {
  System.out.println(i);
}
```


La boucle for (pseudo-langage) [à connaître]

- En pseudo-langage, le compteur prend ses valeurs dans l'intervalle fermé [*a*..*b*].
- incrément de 1

```
afficher("les entiers de 1 à 10")
pour i de 1 à 10 faire
  afficher(i)
fin pour
```

- incrément autre que 1

```
afficher("les entiers pairs de 2 à 10")
pour i de 2 à 10 par pas de 2 faire
  afficher(i)
fin pour
```


La boucle pour chaque (foreach) [pour information]

- Dans une boucle foreach, le compteur de boucle prend ses valeurs successives dans une liste spécifiée dans la boucle. Ces valeurs ne sont pas forcément des entiers.
- en Python

```
for path in ["/bin","/usr/bin"]:
  print(path);
```

- en Bash

```
for path in "/bin" "/usr/bin" ; do
  echo $path
done
```

- en C++

```
for(auto path: {"/bin","/usr/bin"})
  cout << path;
```


La boucle tant que (while)

- Quant on ne sait pas au début de la boucle combien d'itérations vont être faites, on doit utiliser une boucle while ou une boucle do...while.
- Dans une boucle while, les instructions sont répétées tant que la condition indiquée dans la boucle est vraie.
- Dans une boucle while, la condition est testée **avant** l'exécution du corps de la boucle. Si la condition est fausse au démarrage, les instructions ne sont jamais exécutées

La boucle while : responsabilité du programmeur

- Le programmeur doit garantir que la condition de la boucle aura une valeur connue au démarrage de la boucle (initialisation)
- Le programmeur doit garantir qu'à un moment donné, la condition deviendra fausse pour arrêter la boucle (sinon on a une boucle infinie)
- Le programmeur doit garantir que la condition qu'il écrit traduit fidèlement ce qu'il faut faire (ne pas confondre and, or, not et penser à tous les cas)

La boucle while (Java) [à connaître]

- En Java, la boucle while prend la forme suivante

```
while ( condition) {
  instruction1
  instruction2
  ...
}
```

- Exemple :

```
// transférer tout le tas 1 vers le tas 2
while (!tasVide(1)) {
  deplacer(1,2);
}
```


La boucle while (pseudo-langage) [à connaître]

- En pseudo-langage, la boucle while prend la forme suivante

```
tant que condition faire
  instruction1
  instruction2
  ...
fin tant que
```

- Exemple :

```
// transférer tout le tas 1 vers le tas 2
tant que non tasVide(1) faire
  deplacer(1,2)
fin tant que
```


La boucle do...while

- Quant on ne sait pas au début de la boucle combien d'itérations vont être faites, on doit utiliser une boucle while ou une boucle do...while.
- Dans une boucle do...while, les instructions sont répétées tant que la condition indiquée dans la boucle est vraie.
- Dans une boucle do...while, la condition est testée **après** l'exécution du corps de la boucle.
- Il en résulte que les instructions de la boucle seront toujours exécutées au moins une fois.

La boucle do...while : responsabilité du programmeur

- Le programmeur doit garantir que la condition de la boucle aura une valeur connue à la fin du corps de boucle
- Le programmeur doit garantir qu'à un moment donné, la condition deviendra fausse pour arrêter la boucle (sinon on a une boucle infinie)
- Le programmeur doit garantir que la condition qu'il écrit traduit fidèlement ce qu'il faut faire (ne pas confondre and, or, not et penser à tous les cas)

La boucle do...while (Java)

- En Java, la boucle do...while prend la forme suivante

```
do {
  instruction1;
  instruction2;
  ...
} while ( condition);
```

- Exemple :

```
// lire une note
float note;
do {
  System.out.println("Entrez une note (entre 0 et 20)");
  note=in.nextFloat();
} while (note < 0 || note > 20);
```


La boucle do...while (pseudo-langage) [à connaître]

- En pseudo-langage, la boucle do...while prend la forme suivante

```
faire
  instruction1
  instruction2
  ...
tant que condition
```

- Exemple :

```
# lire une note
faire
  afficher("Entrez une note (entre 0 et 20)")
  note ← lireReel()
tant que note<0 ou note>20
```


Choisir la bonne boucle

- Si, avant de commencer la boucle, on sait combien d'itérations on doit faire, il faut utiliser une boucle **for**
- Sinon :
  - Si on sait avant le corps de la boucle si on doit exécuter les instructions ou pas, on utilise une boucle **while**.
  - Sinon, on utilise une boucle **do...while** (parce qu'on saura à la fin du corps de boucle si on doit effectuer une autre itération ou pas).

Choisir la bonne boucle (version alternative)

- Si, avant de commencer la boucle, on sait combien d'itérations on doit faire, il faut utiliser une boucle **for**
- Sinon :
  - Si le corps de la boucle doit être exécuté au moins une fois, on utilise une boucle **do...while**
  - Sinon, on utilise une boucle **while**.

La structure conditionnelle si

- Souvent, on ne veut exécuter des instructions que lorsqu'une condition est satisfaite (vraie).
- Pour cela, on utilise la structure conditionnelle *si*.

La structure conditionnelle if (Java)

- En Java, le if prend la forme suivante

```
if ( condition) {
  instruction1;
  instruction2;
  ...
}
```

- Exemple :

```
// transférer une carte du tas 1 vers le tas 2
if (!tasVide(1)) {
  deplacer(1,2);
}
```


La structure conditionnelle if (pseudo-langage) [à connaître]

- En pseudo-langage, le if prend la forme suivante

```
si condition alors
  instruction1
  instruction2
  ...
fin si
```

- Exemple :

```
// transférer une carte du tas 1 vers le tas 2
si non tasVide(1) alors
  deplacer(1,2)
fin si
```


La structure conditionnelle si...sinon

- Souvent, on ne veut exécuter certaines instructions lorsqu'une condition est satisfaite (vraie) et d'autres instructions lorsque cette même condition est fausse.
- Pour cela, on utilise la structure conditionnelle *si...sinon*.

La structure conditionnelle if...else (Java)

- En Java, le if...else prend la forme suivante

```
if ( condition) {
  // à exécuter quand la condition est vraie
  instruction1
  instruction2
}
else {
  // à exécuter quand la condition est fausse
  instruction3
  instruction4
}
```

- Exemple :

```
if (nbNotes!=0) {
  System.out.println("Moyenne="+(sommeNotes/nbNotes));
}
else {
  System.out.println("Pas de moyenne (aucune note)");
}
```


La structure conditionnelle if...else (pseudo-langage) [à connaître]

- En pseudo-langage, le if...else prend la forme suivante

```
si condition alors
  // à exécuter quand la condition est vraie
  instruction1
  instruction2
sinon
  // à exécuter quand la condition est fausse
  instruction3
  instruction4
fin si
```

- Exemple :

```
si nbNotes≠0 alors
  afficher("Moyenne=",sommeNotes/nbNotes)
sinon
  afficher("Pas de moyenne (aucune note)")
fin si
```

Plus de deux cas ?

- Le if...else ne permet de distinguer que 2 cas différents.
- S'il y a plus que 2 cas à distinguer, il suffit d'imbriquer des if. Il faut alors être très méthodique !
- Exemple :

```
if (a == b) {
  System.out.println("Les nombres sont égaux");
}
else {
  if (a < b) {
    System.out.println("a est plus petit que b");
  }
  else {
    System.out.println("a est plus grand que b");
  }
}
```


Évitez les maladresses

- N'oubliez jamais que dans la partie "else", la condition est nécessairement fausse.
- Exemple :

```
if (a == b) {
  ...
}
else {
  if (a != b && b-a > 3) {
    // ici la condition a!=b est toujours vraie,
    //donc ce test est inutile
    ...
  }
}
```


Évitez les maladresses

- Ne faites pas plusieurs fois le même test...
- Exemple :

```
if (a == b) {
  seq1;
}

if (a != b) {
  seq2;
}
```

doit être remplacé par

```
if (a == b) {
  seq1;
}
else {
  seq2;
}
```


Évitez les maladresses

- Un booléen est déjà vrai ou faux. Inutile de le vérifier une seconde fois.
- Exemple :

```
if (tasVide(1) == true) {
  seq1;
}
```

doit être remplacé par

```
if (tasVide(1)) {
  seq1;
}
```

- Exemple :

```
if (tasVide(1) == false) {
  seq1;
}
```

doit être remplacé par

```
if (!tasVide(1)) {
  seq1;
}
```


Évitez les maladresses

- Un booléen peut être rangé directement dans une variable.
- Exemple :

```
if (a > b) { plusGrand=true; }
else { plusGrand=false; }
```

doit être remplacé par

```
plusGrand = a > b;
```

- Exemple :

```
if (a >= b) { plusPetit=false; }
else { plusPetit=true; }
```

doit être remplacé par

```
plusPetit = ! (a >= b);
// ou mieux encore dans ce cas
plusPetit = a < b;
```


Beaucoup de cas ?

- Quand il y a beaucoup (≥ 3) de cas à traiter, on peut utiliser un switch
- L'action du switch est piloté par une expression (un calcul)
- Ensuite, pour chaque valeur (ou ensemble de valeurs) possible(s) pour l'expression, on détaille les instructions à exécuter (les cas possibles).
- Il y a aussi un cas par défaut qui est exécuté quand la valeur de l'expression ne correspond à aucun cas listé

Le switch en Java

- Exemple en Java :

```
switch(jourSemaine)
{
case 1: // quand jourSemaine==1
  System.out.println("Lundi");
  break; // marque la fin du cas
         // (la fin des instructions)
case 2: // quand jourSemaine==2
  System.out.println("Mardi");
  break;
...
case 6:
case 7: // quand jourSemaine==6 ou jourSemaine==7
  System.out.println("Week-end");
  break;
default:
  System.out.println("Erreur : pas un jour de la
                      semaine");
}
```


Le switch en Bash

- Exemple en Bash :

```
case $jourSemaine in
  1) # quand jourSemaine==1
    echo "Lundi"
    ;; # marque la fin du cas
  2) # quand jourSemaine==2
    echo "Mardi"
    ;;
...
  6|7) # quand jourSemaine==6 ou jourSemaine==7
    echo "Week-end"
    ;;
  *) # par défaut
    echo "Erreur"
    ;;
esac
```


Logique

Conditions, Booléens, Opérateurs

- Une condition utilisée dans un *if* ou un *while* doit donner un résultat qui est soit vrai, soit faux.
- Une variable (ou une expression) qui ne peut prendre que deux valeurs possibles *vrai* et *faux* est appelée booléenne.
- Les opérations possibles sur des booléens sont :
  - le NON logique (not) aussi appelé négation,
  - le ET logique (and) (conjonction),
  - le OU logique (or) aussi appelé ou inclusif (disjonction),
  - le OU exclusif (xor),
  - l'implication,
  - l'équivalence.
- Le sens de ces opérations est défini par une table de vérité, qui donne pour chaque combinaison possible de booléens la valeur du résultat.
- On pourra aussi représenter faux par la valeur 0 et vrai par la valeur 1.

NON logique (not)

- (not A) est vrai lorsque A est faux et inversement.

a	not a
0	1
1	0

ET logique (and)

- (A and B) est vrai si et seulement si A est vrai ainsi que B.
- Dès que l'un des deux est faux, (A and B) est faux.

a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

OU logique (or)

- (A or B) est faux si et seulement si A est faux ainsi que B.
- Dès que l'un des deux est vrai (éventuellement les deux), (A or B) est vrai.

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

OU exclusif (xor)

- (A xor B) est faux si et seulement si A a la même valeur que B.
- Dès que l'un des deux est vrai tandis que l'autre est faux, (A xor B) est vrai.
- Autrement dit, pour que A xor B soit vrai, il faut que l'un des deux soit vrai, mais pas les deux en même temps.

a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0

L'implication

- (A $\Rightarrow$ B) est faux si et seulement si A est vrai et B est faux.
- Quand (A $\Rightarrow$ B) est vrai, alors chaque fois que A est vrai, B doit nécessairement être vrai (et si A est faux, B peut prendre n'importe quelle valeur).

a	b	a $\Rightarrow$ b
0	0	1
0	1	1
1	0	0
1	1	1

L'équivalence

- (A $\Leftrightarrow$ B) est vrai si et seulement si A et B ont la même valeur.

a	b	a $\Leftrightarrow$ b
0	0	1
0	1	0
1	0	0
1	1	1

Notation Java/C/C++

- *and* se note &&
- *or* se note ||
- *not* se note !

Propriétes

- Commutativité
  - a and b = b and a
  - a or b = b or a
  - a xor b = b xor a
- Associativité
  - (a and b) and c = a and (b and c)
  - (a or b) or c = a or (b or c)
  - (a xor b) xor c = a xor (b xor c)
- Distributivité
  - (a and b) or c = (a or c) and (b or c)
  - (a or b) and c = (a and c) or (b and c)
- IdemPotence
  - a or a = a
  - a and a = a

Propriétés

- Élément neutre
  - a and 1 = a
  - a or 0 = a
  - a xor 0 = a
- Élément absorbant
  - a and 0 = 0
  - a or 1 = 1
- Négation
  - a and (not a) = 0
  - a or (not a) = 1
  - a xor (not a) = 1
- Double négation
  - not(not a) = a

Propriétés

- Lois de De Morgan
  - not (a and b) = (not a) or (not b)
  - not (a or b) = (not a) and (not b)
- Ou exclusif
  - a xor b = (a and not b) or (b and not a)
    = (a or b) and (not a or not b)
  - a xor 1 = not a
- Implication
  - a $\Rightarrow$ b = (not a) or b
- Équivalence
  - a $\Leftrightarrow$ b = (a $\Rightarrow$ b) and (b $\Rightarrow$ a)
    = ((not a) or b) and ((not b) or a)
    = (a and b) or ((not a) and (not b))

Exemple de simplification

(not a or b) and a and not(a and b)

(not a or b) and a and ((not a) or (not b))

(not a or (b and not b)) and a

(not a or 0) and a

(not a) and a

0

Évaluation complète/partielle

- Il existe deux manières de calculer la valeur d'une condition
- évaluation complète :
  - on calcule la valeur vérité de chacun des éléments de l'expression, puis on applique les tables de vérité
  - Exemple : n !=0 and 1/n==3 provoquera une erreur si n est égal à 0
- évaluation partielle :
  - Pour le *and* et le *or*, on calcule d'abord la valeur du terme de gauche. Si c'est l'élément absorbant de l'opération, on obtient directement de résultat final sans avoir à calculer la valeur du terme de droite
  - Exemple : n !=0 and 1/n==3 ne provoquera pas d'erreur si n est égal à 0
  - **Attention :** avec une évaluation partielle, les opérateurs logiques ne sont plus commutatifs !

Évaluation partielle

- L'évaluation partielle permet d'écrire des conditions plus courtes, et accélère aussi les calculs
- Quand un test risque de provoquer une erreur, il faut toujours vérifier **au préalable (donc avant le test problématique)** que cette erreur ne se produira pas :
  - if n !=0 and 1/n==3 : ...
  - if not tasVide(1) and sommetCoeur(1) : ... pour vérifier qu'il y a un cœur
  - if tasVide(1) or not sommetCoeur(1) : ... pour vérifier qu'il n'y a pas de cœur
- Sans évaluation partielle, il faut décomposer les tests :

```
if ( n != 0)
  if ( 1/n == 3 )
    ...
```

- La plupart des langages (dont Python, Java, C, C++) utilisent une évaluation partielle.

Variables, Types, Affectation, Expressions

Variable

- Une variable contient une information d'un certain type.
- La valeur d'une variable est l'information contenue dans la variable.
- Le contenu d'une variable (sa valeur) peut être modifié à volonté
- Dans un calcul (une expression), le nom d'une variable est automatiquement remplacé par sa valeur
- Une variable doit avoir un nom. En général, ce nom doit commencer par une lettre. On peut ensuite trouver des lettres, ou des chiffres, ou des soulignés.
- De nombreux langages distinguent majuscules et minuscules dans les noms de variable
- Une variable est l'équivalent d'un récipient en cuisine. Chaque récipient (casserole, bouteille, boîte à œufs, ...) est adapté à un type de contenu spécifique.

Type

- Le type d'une variable indique la nature de l'information contenue dans la variable. Il indique quelles sont la ou les informations contenues, comment elle sont représentées en mémoire (cf. R1.03), et comment ces informations peuvent être manipulées (les opérations possibles).
- Les types de base possibles sont :
  - booléen : contient soit faux, soit vrai (0 ou 1)
  - entier : un entier représenté sur un nombre limité de bits (par exemple 32 ou 64 bits)
  - réel : un nombre réel représenté de manière approximative (donc les calculs sur les réels seront souvent faux !)
  - caractère : un seul caractère considéré isolément
  - une chaîne de caractères : une succession de caractères
- Ces types de base permettent de créer de nouveaux types (tableaux, classes, ...)

Déclaration, Initialisation

- En Java, on doit d'abord déclarer une variable, c'est à dire demander sa création en précisant son nom et son type. Exemple en Java :

```
int n; // déclare une variable n de type entier (integer)
```

- Une fois créée, on ne peut plus modifier le type de la variable.
- Une fois la variable créée, il faut lui donner une valeur avant de pouvoir l'utiliser. C'est l'initialisation. Exemple :

```
n=0; // donner une valeur à la variable
```

- On peut déclarer et initialiser en même temps : `int n=0;`
- Une variable qui n'est pas initialisée peut contenir une valeur aléatoire ou incohérente. **On ne doit jamais utiliser une variable qui n'est pas initialisée.**
- Dans une expression, le nom d'une variable est remplacé par sa valeur. Exemple :

```
System.out.println(n); // afficher la valeur de la variable n
System.out.println(n*n+2*n+1);
```


Déclaration des variables

- Les variables qu'on utilise uniquement dans une fonction doivent être déclarées au début de cette fonction. Elle ne sont accessibles que dans cette fonction.
- Les variables qu'on veut utiliser dans plusieurs fonctions doivent être déclarées au début de la classe de votre programme. Cela simule des variables globales.
- On reviendra sur ces notions (et la notion de portée) quand on vous présentera les fonctions.
- L'obligation de déclarer ses variables est très utile. Cela permet de détecter les erreurs dans le nom des variables qu'on utilise. Exemple :

```
int maVariable;
maVariable = 0; // OK
mavariable = 1; // ERREUR ! variable inconnue
maVarable = maVariable + 1; // ERREUR !
```


Affectation en pseudo-langage

- Pour changer la valeur d'une variable, on utilise une opération qui se nomme affectation.
- En pseudo langage, le symbole de l'affectation est la flèche vers la gauche ←. Ce symbole se lit "reçoit la valeur".
- À gauche du symbole d'affectation (←), on doit toujours trouver un nom de variable.
- À droite du symbole d'affectation (←), on doit toujours trouver une expression qui représente un calcul à effectuer. La valeur résultant de ce calcul sera rangée dans la variable.
- On calcule d'abord la valeur de l'expression à la droite de ← et ensuite seulement on range cette valeur dans la variable désignée à la gauche de ←
- On dit qu'on assigne ou qu'on affecte une valeur à une variable.
- Exemples :

```
v ← 0
v ← v+1
```


Affectation

- En Java, C, C++, le symbole de l'affectation est le signe = à la place de ←, mais le mécanisme reste le même. = doit toujours être lu comme "reçoit la valeur"
- En aucun cas ce symbole = ne doit être compris comme une équation mathématique !
- À gauche du symbole d'affectation (=), on doit toujours trouver un nom de variable.
- À droite du symbole d'affectation (=), on doit toujours trouver une expression qui représente un calcul à effectuer. La valeur résultant de ce calcul sera rangée dans la variable.
- On calcule d'abord la valeur de l'expression à la droite de = et ensuite seulement on range cette valeur dans la variable désignée à la gauche de =
- Exemples :

```
v = 0;
v = v + 1;
```

Affectation

- Une variable ne peut contenir qu'une valeur à la fois.
- Quand on assigne/affecte une nouvelle valeur à une variable, la valeur précédente est remplacée par la nouvelle valeur.
- De ce fait, l'ancienne valeur est perdue !

Expression

- Une expression représente un calcul mathématique à effectuer.
- On peut trouver une expression à la droite du symbole d'affectation, ou dans des appels à des fonctions.
- Dans une expression, un nom de variable est systématiquement remplacé par sa valeur.
- Les opérations mathématiques à effectuer sont représentées par des opérateurs.
- Un opérateur effectue ses calculs sur des opérandes. Par exemple, dans $a + b$, l'opérateur est le $+$ et les deux opérandes sont a et b.

Opérateurs courants

- Opérateurs relationnels : ils permettent de comparer deux valeurs et retournent soit vrai, soit faux
  == (test d'égalité), != (différence), <, <=, >, >=
- Opérateurs arithmétiques : +, -, *, / (division réelle), / (division entière), % (modulo)
- Opérateurs bit à bit : & (et bit à bit), | (ou bit à bit), ^ (xor bit à bit), ~ (négation bit à bit, complément à 1), << décalage à gauche, >> décalage à droite.
- La conversion de type (cast) : convertir une valeur en un autre type (quand cela a du sens)
- ...

Opérateurs relationnels

- Ils permettent de comparer 2 valeurs.
- Les deux termes de chaque côté de l'opérateur doivent être **comparables**. Ils doivent avoir le même type ou il doit y avoir une conversion évidente, et la comparaison doit avoir du sens. Exemples : 1==2 (OK), 1==2.0 (OK), 1=="1" (NON !), rouge<bleu (NON !)
- test d'égalité : noté == dans de nombreux langages
- test de différence : noté != (pas égal) (ou parfois <> comme en SQL)
- comparaisons : <, <=, >, >=
- On rappelle que `not (x==y)` se simplifie en `x!=y`, que `not (x!=y)` se simplifie en `x==y`, que `not (x<y)` se simplifie en `x>=y`, que `not (x<=y)` se simplifie en `x>y`, ...
- En pseudo-langage, on utilise les notations mathématiques habituelles : $=, \neq, \leq, \geq, <, >$

= et ==

- Attention ! En Java/C/C++, l'opérateur d'affectation se note par un = et le test d'égalité se note par un double signe égal (= =) l'un à la suite de l'autre. Il ne faut surtout pas confondre les deux.
- L'opérateur d'affectation sert à placer une valeur dans une variable. À sa gauche, on doit trouver le nom d'une variable, à sa droite, l'expression calculant la valeur à placer dans la variable. L'opérateur d'affectation ne renvoie pas de résultat. Il constitue une instruction à part entière.
- Le test d'égalité sert à comparer deux valeurs. On doit trouver de chaque côté une expression calculant les valeurs à comparer. Le test d'égalité renvoie un résultat qui est soit vrai, soit faux. Le test d'égalité n'est pas une instruction

Opérateurs arithmétiques

- Ils permettent d'effectuer des calculs sur des entiers ou des réels : +, -, *, /
- Attention : dans la plupart des langages, il y a des limites sur les plus grands (plus petits) nombres qu'on peut représenter. Quand on dépasse ces limites, les calculs sont faux ! Par exemple, sur 16 bits non signés, on peut obtenir 65535+1=0, ou bien en signé, 32767+1=-32768 (cf. R1.03).
- Attention : les calculs avec des réels sont très souvent faux ! Par exemple (1/3)*3 !=1, 0.1*10 !=1, ...

Division entière, division réelle

- La division entière d'un nombre a (dividende) par un nombre b (diviseur) produit un quotient q et un reste r tel que $a = b * q + r$ avec $r < b$.
- En C/C++/Java, quand la division porte sur des entiers, c'est une division entière : 10/6==1
- Le reste de la division entière de a par b se note a%b (a modulo b).
- Par définition, ce reste est toujours compris entre 0 et $b - 1$.
- Exemple 10%6==4
- En C/C++/Java, quand la division porte sur au moins un nombre réel, c'est une division réelle : 10/4.0==2.5, 10.0/4==2.5, 10.0/4.0==2.5

Utilisation du modulo

- Le modulo (%) est très utile pour gérer un phénomène périodique.
- Quand on a un phénomène périodique de période n, on gagne souvent à utiliser un modulo n.
- Il ne faut pas oublier que modulo n produit un résultat compris entre 0 et $n - 1$
- De ce fait, il faut d'abord se ramener à un codage compris entre 0 et $n - 1$, puis utiliser un modulo n, puis se ramener au codage de départ.
- Exemple : si m est le mois courant, quel mois serons nous dans x mois ?
  Réponse : $((m - 1 + x)\%12) + 1$

Conversions de type

- Un conversion de type permet de changer la représentation interne d'une valeur.
- Ce changement de réprésentation peut induire un changement de la valeur (conversion d'un réel en entier par exemple)
- Pour traduite une valeur v en un type T, on écrit (T)v.
- Les conversions courantes dans tous les langages :
  - float $\longrightarrow$ int
    Le plus souvent la conversion se fait par troncature, c'est à dire en effaçant les chiffres après la virgule. Exemple : (int)1.1==1, (int)1.6==1
    Pour arrondir un nombre x à partir d'une troncature, on peut écrire (int)(x+0.5)
  - int $\longrightarrow$ float
    Exemple : (float)5==5.0

Priorité des opérateurs

- Vous savez que la multiplication a priorité sur l'addition, ce qui signifie que a * b + c doit être compris comme (a * b) + c.
- En l'absence de parenthèses, un opérateur prioritaire doit être calculé avant un opérateur moins prioritaire. Cela permet d'éviter des parenthèses et d'alléger la lecture.
- D'un langage à l'autre, les priorités peuvent être différentes.
- Si on ne connait pas bien les priorités du langage, il vaut mieux être prudent et ajouter des parenthèses.

Opérations bit à bit

- Comme leur nom l'indique, les opérations bit à bit opèrent sur chacun des bits d'un ou plusieurs nombres entiers.
- Les opérandes sont donc des entiers, et les opérations bit à bit effectuent leur calcul sur chacun des bits de la représentation binaire de ces nombres.
- Pour effectuer les calculs, il faut donc d'abord convertir les entiers en binaire (on notera $(b_{n-1}, ...b_2, b_1, b_0)_2$ pour un nombre b sur n bits), puis effectuer l'opération bit à bit. Le résultat est toujours un entier.
- En Python, C, C++ et Java, ces opérations se notent comme suit :

$\sim expr$	complément à 1 (négation bit à bit)
$expr << expr$	décalage à gauche
$expr >> expr$	décalage à droite
$expr$ & $expr$	ET bit à bit
$expr \hat{} expr$	OU exclusif bit à bit
$expr \mid expr$	OU bit à bit

Opérations logiques bit à bit

Un bit à 0 représente FAUX, un bit à 1 représente VRAI.

- complément à 1 (négation bit à bit) : $r = \sim a$
  $\forall i, r_i = not(a_i)$
  exemple : ∼0xA vaut 0x5 (complément à 1 de 1010=0101)
- et bit à bit : $r = a \,\&\, b$
  $\forall i, r_i = and(a_i, b_i)$
  exemple : 0xA & 0x6 vaut 0x2 (1010 ET bit à bit 0110=0010)
- ou bit à bit : $r = a \mid b$
  $\forall i, r_i = or(a_i, b_i)$
  exemple : 0xA | 0x6 vaut 0xE (1010 OU bit à bit 0110=1110)
- ou exclusif bit à bit : $r = a \hat{} b$
  $\forall i, r_i = xor(a_i, b_i)$
  exemple : 0xA ^ 0x6 vaut 0xC (1010 XOR bit à bit 0110=1100)

Décalages

- décalage à gauche $r = a << d$
  on décale les bits de d rangs vers la gauche, ce qui revient à supprimer les d bits de poids fort (à gauche) du nombre et insérer d bits à 0 en poids faible (à droite)
  exemple sur 8 bits :
  $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)_2 << 2 = (a_5, a_4, a_3, a_2, a_1, a_0, 0, 0)_2$
  Si l'on ne perd pas de bit significatif, cette opération est une multiplication par 2^d. De manière générale, $a << d = (a.2^d) \mod 2^n$ où n est le nombre de bits utilisés pour représenter les nombres.
- décalage à droite $r = a >> d$
  on décale les bits de d rangs vers la droite, ce qui revient à supprimer les d bits de poids faible (à droite) du nombre et insérer d bits à 0 en poids fort (à gauche)
  exemple sur 8 bits :
  $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)_2 >> 3 = (0, 0, 0, a_7, a_6, a_5, a_4, a_3)_2$
  Cette opération est une division entière par 2^d.

Utilisation des opérations logiques bit à bit

- Quand on veut forcer le bit d'indice i à prendre la valeur 1 dans un nombre x, on fait un OU bit à bit avec un nombre m (masque) dont tous les bits sont à 0 sauf le bit d'indice i qui est à 1. Ce nombre m s'obtient pas $1 << i$
- Quand on veut forcer le bit d'indice i à prendre la valeur 0 dans un nombre x, on fait un ET bit à bit avec un nombre m (masque) dont tous les bits sont à 1 sauf le bit d'indice i qui est à 0. Ce nombre m s'obtient par $\sim (1 << i)$.
- Quand on veut inverser la valeur du bit d'indice i dans un nombre x, on fait un XOR bit à bit avec un nombre m (masque) dont tous les bits sont à 0 sauf le bit d'indice i qui est à 1. Ce nombre m s'obtient pas $1 << i$
- On peut bien sûr généraliser et modifier simultanément la valeur de plusieurs bits.

Fonctions mathématiques

En écrivant `import java.lang.Math;` au début du programme, on a accès à de nombreuses fonctions mathématiques usuelles :

- Math.sqrt(x)
- Math.cos(x)
- Math.log(x)
- Math.log10(x)
- Math.exp(x)
- Math.PI
- Math.E
- ...

Expression

- Une expression représente un résultat calculé, une valeur qu'il faut utiliser en la rangeant dans une variable, en l'affichant ou en la passant à une procédure ou fonction.
- Cela n'a pas de sens de calculer un résultat pour le laisser tomber aussitôt. Il faut donc toujours utiliser la valeur calculée (en la rangeant dans une variable, en l'affichant ou en la passant à une procédure ou fonction).

Différence Expression/Instruction

- Une instruction est un ordre donné à la machine. L'instruction va permettre de réaliser une certaine action.
- Il ne faut pas confondre instruction et expression. Une expression représente une certaine valeur (calculée) tandis qu'une instruction ne représente aucune valeur.

Affichage

- En Java, pour afficher des résultats, on utilise la fonction System.out.println
- On liste entre parenthèses et en les séparant par des +, les différents élements que l'on veut affficher (toujours des expressions)
- Exemple : `System.out.println("1+1="+(1+1)+"en base 10");`
- System.out.println passe à la ligne après avoir fait l'affichage
- System.out.print ne passe pas à la ligne après avoir fait l'affichage
- Exemple :

```
for( int i = 0; i < 10 ; i++) {
  System.out.print(' ');
}
System.out.println();
```


Lecture au clavier

- En Java, pour lire des données au clavier, on utilisera le squelette suivant :

```
import java.io.*;
import java.util.*;

class NomDuProgramme {
  // les variables "globales"
  Scanner in=new Scanner(System.in);

  void run() {
    // le code de votre programme
    int n=in.nextInt();
    float f=in.nextFloat();
    String line=in.nextLine();
  }
  // lancer run() en évitant ''au mieux'' les objets
  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
}
```


Les constantes nommées

- Une constante est un nom que l'on donne à une valeur. Contrairement à une variable, une constante ne peut pas changer de valeur au cours de l'exécution du programme.
- Chaque fois qu'on utiliser une même valeur, avec la même signification, à différents endroits du programme, il faut penser à créer une constante.
- Si on ne le fait pas, le jour où l'on doit changer cette valeur, il devient très difficile de savoir où faire le changement dans le programme
- Exemple en Java : `final double tva=20;`

Les constantes littérales

Le type d'une variable régit aussi comment noter les valeurs qu'on peut assigner à ces variables :

- boolean : false, true
- int : 1234 (base 10), 0234 (base 8), 0x234F (base 16)
- float, double : 36, 1.5, 1.5E4, -2.3E-10
- char : 'A', 'e', '\\n'
- String : "une chaîne"

Introduction aux procédures

Introduction aux procédures

- Dans un programme, il arrive souvent qu'on doivent exécuter les mêmes instructions à différents endroits du programme.
- Pour éviter ces répétitions, les bugs de copier/coller et les problèmes de mise à jour, on donne un nom à la séquence d'instructions, et chaque fois qu'on veut les exécuter, on invoque ce nom.
- L'association d'un nom et d'une séquence d'instructions s'appelle une procédure.
- On commence par **définir** une procédure (associer le nom et le code). En Java/C/C++, , la définition d'une procédure commence toujours par le mot clef *void*, suivi du nom de la procédure, suivi de parenthèses et du code de la procédure (entre accolades).
- Ensuite, on peut **appeler** la procédure (exécuter les instructions associées au nom). En Java/C/C++, , l'appel d'une procédure commence par le nom de la procédure, suivi de parenthèses et se termine par un point-virgule.
- L'appel d'une procédure est une instruction.
- Les procédures sont un cas particulier de fonctions (vues dans la seconde moitié du semestre), et de méthodes (vues au second semestre).

Exemple de procédure (1)

```
import java.io.*;

class NomDuProgramme {
  // définition de la procédure
  void bonjour() {
    System.out.println("Bonjour !");
  }
  void run() {
    bonjour(); // 1er appel
    bonjour(); // 2ème appel
  }

  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
}
```


Variables locales/"globales"

- Une procédure peut avoir ses propres variables. On les appelle des **variables locales**.
- On les définit au début de la procédure. Elle ne peuvent être utilisées que dans la procédure où elle sont définies. Elles sont détruites à la fin de l'exécution de la procédure.
- Les variables qui ne sont définies dans aucune procédure sont des variables "globales". Elles peuvent être utilisées dans toutes les procédures. Il faut manipuler ces variables avec précautions : toutes les procédures qui les utilisent doivent le faire de manière cohérente.

Exemple de variables locales/"globales"

```
class NomDuProgramme {
  // les variables "globales"
  Scanner in = new Scanner(System.in);

  // définition de la procédure
  void equationSecondDegre() {
    // les variables locales
    double a, b, c; // coefficients de l'équation
    System.out.println("Entrez a : )";
    a = in.nextDouble();
    ...
  }
  void run() {
    equationSecondDegre(); // appel/exécution de la procédure
  }

  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
```


Exemple de procédure (2)

```
import java.io.*;
import cartes.*;

class NomDuProgramme extends Cartes {
  // définition de la procédure
  void transfererTas1Vers2() {
    while ( ! tasVide(1) ) {
      deplacer(1, 2);
    }
  }
  void run() {
    transfererTas1Vers2(); // appel
  }

  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
}
```


Paramètres des procédures

- On voit sur l'exemple précédent qu'il manque un mécanisme pour transmettre une (ou des) information(s) aux procédures, comme par exemple le numéro des tas que l'on veut manipuler. Ce mécanisme est le passage de paramètres.
- Quand on définit une fonction, on ne sait pas quelle information nous sera transmise. On lui donne donc un nom qui n'est utilisé qu'à l'intérieur de la définition. Il faut aussi définir le type de l'information transmise. On parle de *paramètre formel* (*parameter* en anglais) car si on change le nom utilisé dans la définition de la procédure, le sens du programme ne changera pas.
- Les paramètres formels se comportent comme des variables locales à la procédure.
- Quand on appelle une fonction, on sait parfaitement quelles informations doivent être transmises. Ces informations transmises sont appelées *paramètres effectifs* (*argument* en anglais). Si on change la valeur de ces paramètres effectifs, le sens du programme va sans doute changer.
- On met les paramètres (formels ou effectifs) entre les parenthèses qui suivent le nom de la procédure, en les séparant par des virgules.

Passage de paramètres par valeur

Dans le mode de passage par valeur des paramètres (le seul que nous utiliserons pour l'instant), voici ce qui se passe lors de l'appel d'une procédure :

- Avant d'exécuter les instructions de la procédure, le 1er paramètre effectif est copié dans le 1er paramètre formel (qui se comporte comme une variable locale), le 2ème paramètre effectif est copié dans le 2ème paramètre formel, et ainsi de suite.
- Si l'on modifie la valeur d'un paramètre formel dans la procédure, cela ne changera rien à la valeur du paramètre effectif.

Exemple de paramètres (1)

```
class NomDuProgramme extends Cartes {
  // définition de la procédure
  // utilisation de 2 paramètres formels
  void transfererTas(int source, int dest) {
    while ( ! tasVide(source) ) {
      deplacer(source, dest);
    }
  }
  void run() {
    // appels de la procédure avec des paramètres effectifs
    transfererTas(1, 2); // appel avec source==1, dest==2
    transfererTas(2, 3); // appel avec source==2, dest==3
    transfererTas(3, 1); // appel avec source==3, dest==1
  }

  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
}
```

Exemple de paramètres (2)

```
import java.io.*;

class NomDuProgramme {
  // définition de la procédure
  void void afficherCarre(int n) {
    n = n * n; // modification du paramètre effectif
    System.out.println(n);
  }
  void run() {
    int x = 2;
    int n = 3;
    afficherCarre(4); // pas de modification
    afficherCarre(x); // pas de modification
    afficherCarre(n); // pas de modification
  }

  public static void main(String[] args) {
    new NomDuProgramme().run();
  }
```

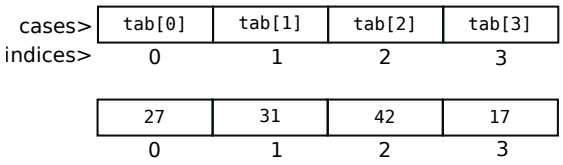

Les tableaux

Les tableaux

- Un tableau est le regroupement de plusieurs variables qui sont **nécessairement** du même type et qui seront normalement utilisées de la même manière.
- Les variables du tableau sont repérées par un numéro appelé **indice** (tableau à une dimension) ou bien par plusieurs indices (tableau à plusieurs dimensions)

Indice dans un tableau

- L'indice utilisé pour identifier les variables qui composent un tableau doit être du type entier.
- Les indices utilisés sont nécessairement consécutifs.
- Dans de nombreux langages, le premier indice d'un tableau est 0.
- De ce fait, si le tableau contient n cases, l'indice de la dernière case est $n - 1$
- Vous ne devez pas confondre l'indice d'une case et le contenu de la case



27	31	42	17
0	1	2	3

Accès aux éléments d'un tableau

- Pour accéder à un élément du tableau, il suffit de donner le nom du tableau et de mettre entre crochets l'indice de l'élément que l'on veut manipuler : `nom_tableau[indice]`
- `nom_tableau[indice]` désigne la case d'indice `indice` dans le tableau. Cette case est une variable que l'on peut utiliser comme toute autre variable.
- Pour un tableau contenant n cases, l'indice doit impérativement être compris entre 0 et $n - 1$. Si ce n'est pas le cas, votre programme plante.

Exemple en Java

```
//une constante égale à 10
final int N=10;

// déclaration d'un tableau de 10 entiers
int[] tab = new int[N];

// initialisation des cases
for(int i = 0; i < N; i++) {
  tab[i] = 5 * i;
}

// modification d'une case
tab[2] = (tab[1] + tab[3]) / 2;

// affichage 1ère et dernière case
System.out.println(tab[0] + " " + tab[N-1]);
```


Taille du tableau

- Un tableau a une taille fixe.
- Une fois créé, on ne peut plus changer la taille du tableau.
- Si le tableau se révèle trop petit, il faut utiliser des techniques avancées pour créer un nouveau tableau, copier les éléments de l'ancien tableau dans le nouveau, puis supprimer l'ancien tableau. Tout ça a un coût non négligeable et requiert une certaine maîtrise.
- En Java, la taille d'un tableau `tab` peut s'obtenir en écrivant `tab.length`

Boucle sur un tableau

- Pour parcourir tous les éléments d'un tableau de n cases (avec des indices compris entre 0 et $n - 1$, on écrit naturellement :

```
for(int i = 0; i< tab.length; i++) {
  System.out.println(tab[i]);
}
```

- Vos boucles doivent toujours garantir que l'indice utilisé dans le tableau est compris entre 0 et $n - 1$. Exemple :

```
// trouver le premier 0 du tableau
int i = 0;
while (i < tab.length && tab[i] != 0) {
  i++;
}

if (i < tab.length) {
  System.out.println("Le premier 0 est à l'indice " + i);
}
else {
  System.out.println("Pas de 0 dans le tableau");
}
```


Algorithmes de base sur les tableaux

Somme des éléments d'un tableau

```
// somme des éléments d'indice compris dans [debut..fin[
double somme = 0;
for (int i = debut; i < fin; i++) {
  somme = somme + tab[i];
}
```

Plus grand élément

```
// recherche de l'indice du plus grand élément dans
// le sous tableau [debut,fin[
int indiceMax = debut;
double max = tab[indiceMax];
for (int i = debut + 1; i < fin; i++) {
  if (tab[i] > max) {
    max = tab[i];
    indiceMax = i;
  }
}
```


Recherche d'un élément dans un tableau non trié (1)

```
// on cherche l'indice de la première case égale à e
boolean trouve = false;
int i = 0;
while (i < tab.length && ! trouve) {
  if (tab[i] == e) {
    trouve = true;
  }
  else {
    i = i + 1;
  }
}

if (trouve) {
  System.out.println(e + " se trouve dans la case d'indice " + i);
}
else {
  System.out.println(e + " n'est pas dans le tableau");
}
```


Recherche d'un élément dans un tableau non trié (2)

```
int i = 0;
while (i < tab.length && tab[i] != e) {
  i++;
}

if (i < tab.length) {
  System.out.println(e + " se trouve dans la case d'indice " + i);
}
else {
  System.out.println(e + " n'est pas dans le tableau");
}
```


Recherche linéaire dans un tableau trié

```
// recherche dans un sous-tableau [debut..fin[ trié
// par ordre croissant
int i = debut;
while (i < fin && tab[i] < e ) {
  i = i + 1;
}

if (i < fin && tab[i] == e) {
  System.out.println(e + " se trouve dans la case d'indice " + i);
}
else {
  System.out.println(e + " n'est pas dans le tableau");
}
```


Recherche dichotomique dans un tableau trié

```
// recherche par dichotomie dans [debut..fin[
boolean trouve = false;
// tant que l'intervalle n'est pas vide et qu'on n'a pas trouvé
while (debut < fin && !trouve) {
  milieu = (debut + fin) / 2;
  if (tab[milieu] == e) {
    trouve = true;
  } else {
    if (e < tab[milieu]) { fin = milieu; } else { debut = milieu + 1; }
  }
}
if (trouve) {
  System.out.println(e + " se trouve dans la case d'indice " + milieu);
} else {
  System.out.println(e + " n'est pas dans le tableau");
}
```


Affichage des éléments du tableau avec séparateur

```
System.out.print(tab[0]);
for (int i = 0; i < tab.length; i++) {
  System.out.print(", " + tab[i]);
}
System.out.println();
```


Tri par sélection (éléments de type T)

```
for (int i=0; i < tab.length-1; i++) {
  // rechercher le plus petit élément dans [i..N[
  int indiceMin = i;
  T min = tab[indiceMin];
  for (int j = i + 1; j < tab.length; j++) {
    if (tab[j] < min) {
      min = tab[j];
      indiceMin = j;
    }
  }
  // mettre le plus petit élément à la position i
  T tmp = tab[i];
  tab[i] = tab[indiceMin];
  tab[indiceMin] = tmp;
}
```


Tri par insertion (éléments de type T)

```
for (int i = 0 ; i < tab.length; i++) {
  // on doit insérer tab[i] dans [0..i[ qui est déjà trié
  int pos = 0;
  while (pos < i && tab[pos] < tab[i]) {
    pos = pos + 1; // l'élément doit aller plus loin
  }
  // faire de la place
  T tmp = tab[i];
  for (int j = i-1; j >= pos; j--) {
    tab[j+1] = tab[j];
  }
  // insérer à la bonne place
  tab[pos] = tmp;
}
```


Tri à bulle (éléments de type T)

```
boolean estTrie;
do  {
  estTrie = true;

  for (int i = 0; i < tab.length - 1; i++) {
    if (tab[i] > tab[i+1]) {
      // permuter les 2 éléments pour mettre un peu plus d'ordre
      estTrie = false;
      T tmp = tab[i];
      tab[i] = tab[i+1];
      tab[i+1] = tmp;
    }
  }
} while ( !estTrie );
```


Tableau avec un nombre variable d'éléments

- Quand on veut ranger dans un tableau un nombre d'éléments qui peut varier au cours du temps, la technique la plus simple et la plus efficace est de n'utiliser que les *n* premières cases.
- Le tableau doit être créé avec un nombre de cases égal au maximum d'éléments possibles *max*.
- Il faut avoir une variable *n* qui indique combien de cases sont occupées actuellement dans le tableau (toujours les premières cases)