BASES DE LA PROGRAMMATION

STRUCTURES RÉPÉTITIVES

BOUCLES

- On a très souvent besoin de répéter un traitement un grand nombre de fois
- Une **boucle** est une structure permettant à un bloc de code de se répéter
  - cela permet de ne pas copier/coller le même bout de code x fois
  - d'autant plus que, la plupart du temps, on ne sait pas à l'avance *combien de fois* on doit répéter le traitement

INSTRUCTION WHILE (TANT QUE)

- On va répéter un bloc TANT QUE la condition entre parenthèses est vraie

```
TANT QUE (condition est vraie)
    Action A
// suite du programme
```

```
TANT QUE (le feu est rouge)
    Appuyer sur le frein
// suite du programme
```

PROBLÈME

UN EMPLOYÉ GAGNE 15 € DE L'HEURE. DEMANDER LE NOMBRE D'HEURES TRAVAILLÉES DANS LA SEMAINE ET AFFICHER SON SALAIRE. LA VALEUR ENTRÉE DOIT ÊTRE ENTRE 0 ET 42 (REDEMANDER AUTANT DE FOIS QUE NÉCESSAIRE).

ALGORITHME

- Demander le nombre d’heures travaillées
- **TANT QUE** le nombre d’heures est invalide
  - demander le nombre d’heures travaillées
- Calculer le salaire
- Afficher le salaire

WHILE EN JAVA

```java
while (condition) {
  // ce bloc se répète tant que condition est vraie
  // la condition est ré-évaluée à chaque fin de bloc
}
```

IMPLÉMENTATION EN JAVA

EXEMPLE D'IMPLÉMENTATION

```java
public static void main(String[] args) {
  int tauxHoraire = 15;
  int heuresMax = 42;

  // Récupérer le nombre d'heures travaillées
  System.out.print("Combien d'heures travaillées cette semaine ? ");
  Scanner clavier = new Scanner(System.in);
  int heuresTravaillees = clavier.nextInt();

  // Validation de l'entrée
  // On continue de demander TANT QUE l'entrée est invalide
  while (heuresTravaillees < 0 || heuresTravaillees > heuresMax) {
    System.out
        .println("Nombre d'heures invalide. Entrez un nombre entre 0 et " + heuresMax + ": ");
    heuresTravaillees = clavier.nextInt();
  }
  clavier.close();

  // Calcul et affichage
  int salaireBrut = tauxHoraire * heuresTravaillees;
  System.out.println("Salaire brut : " + salaireBrut);
}
```

WHILE - PARTICULARITÉS

- La boucle continue TANT QUE la condition est vraie
  - il faut que la variable testée **soit modifié** dans le bloc répété, sinon on a une **boucle infinie**
- La condition pourrait être fausse dès le départ
  - la condition est testée **au début**
  - donc il est possible que le bloc ne soit **jamais exécuté**
- Utilisé habituellement pour répéter un traitement qui pourrait ou non avaoir besoin d'être exécuté en fonction du contexte

INSTRUCTION FAIRE TANT QUE (DO WHILE)

- Idem que TANT QUE, sauf que la **condition n'est testée qu'à la fin du bloc**
  - le bloc s'exécute donc **au moins une fois**, même si la condition est fausse dès le départ

```
FAIRE
    Action A
TANT QUE (condition est vraie)
// suite du programme
```

```
FAIRE
    Préparer gâteau
    Cuire gâteau
TANT QUE (gâteau brûlé)
// suite du programme
```

PROBLÈME

ON REPREND LE PROBLÈME PRÉCÉDENT

ANALYSE

- On remarque, dans la solution précédente, que l'on demande dans tous les cas un nombre à l'utilisateur (au moins une fois)
- On peut remanier le code avec un FAIRE TANT QUE pour n'écrire le traitement de demande qu'une fois

DO WHILE EN JAVA

```
do {
  // ce bloc sera exécuté au moins une fois,
  // puis répété tant que la condition est vraie
} while (codition);
```

IMPLÉMENTATION EN JAVA

EXEMPLE D'IMPLÉMENTATION

```
public static void main(String[] args) {
  int tauxHoraire = 15;
  int heuresMax = 42;
  int heuresTravaillees;
  Scanner clavier = new Scanner(System.in);
  do {
    System.out
        .print("Combien d'heures travaillées cette semaine (entre 0 et " + heuresMax + ") ? ");
    heuresTravaillees = clavier.nextInt();
  } while (heuresTravaillees < 0 || heuresTravaillees > heuresMax);
  clavier.close();

  int salaireBrut = tauxHoraire * heuresTravaillees;
  System.out.println("Salaire brut : " + salaireBrut);
}
```

DO WHILE - PARTICULARITÉS

- Le bloc s'exécute puis se répète TANT QUE la condition est vraie
  - le bloc est toujours exécuté **au moins une fois**
  - car la condition est testée **à la fin**
  - il faut toujours que la variable testée **soit modifié** dans le bloc répété pour éviter la **boucle infinie**
- Beaucoup plus rarement utilisé que le WHILE, mais pratique lorsque le problème s'y prête

INSTRUCTION POUR (FOR)

- Cette fois le bloc A va se répéter **un nombre de fois spécifié**

```
POUR i de 0 à n
    Action A
// suite du programme
```

```
POUR i de 1 à 10
    Afficher i
// suite du programme
```

INSTRUCTION FOR EN JAVA

```java
for (int i = 1; i <= 10, i++) {
  // <traitement>
  // ce bloc sera ici répété exactement 10 fois
}
```

- Ce code se lit : « **pour i allant de 1 à 10 (inclus) par incrément de 1, faire <traitement>** »
- *incrément de 1* signifie : de 1 en 1
- en effet `i++` dans la phase de mise à jour est équivalent à `i = i + 1`

PROBLÈME

CALCULER ET AFFICHER LE TOTAL D'UN PANIER D'ARTICLES DE PRIX DIFFÉRENTS. ON DEMANDERA À L'AVANCE LE NOMBRE TOTAL D'ARTICLES PUIS, POUR CHAQUE ARTICLE, ON DEMANDERA SON PRIX.

ANALYSE

- On doit remarquer qu'il y a un traitement répété pour chaque article du panier
  - on va demander le prix de l'article pour l'ajouter à un total
- Cela nécessite donc une boucle
- On connaît le nombre de répétition (on le demande à l'avance à l'utilisateur)
  - donc on va utiliser un **for**

ALGORITHME

- Demander le nombre d’articles
- Initialiser le total du panier à 0
- Pour chaque article
  - Demander le prix de l’article
  - Ajouter le prix au total du panier
- Afficher le total du panier

IMPLÉMENTATION EN JAVA

EXEMPLE D'IMPLÉMENTATION

```java
public static void main(String[] args) {
  // Récupération de la quantité d'articles du panier
  System.out.print("Entrez le nombre total d'articles : ");
  Scanner clavier = new Scanner(System.in);
  int quantite = clavier.nextInt();

  // On initialise le total à 0
  // On va accumuler les prix des articles dans ce total
  // au fur et à mesure
  double total = 0;

  // La boucle s'exécute "quantite" fois
  // À chaque itération (répétition), on demande le prix de l'article
  // et on l'ajoute au total
  for (int i = 1; i <= quantite; i++) {
    System.out.print("Entrez le prix de l'article " + i + " : ");
    double prix = clavier.nextDouble();
    total = total + prix;
  }
  clavier.close();
  System.out.println("Le total du panier est : " + total);
}
```

FOR - PARTICULARITÉS

- On utilise une boucle FOR **quand on connaît précisément le nombre de répétitions**
  - souvent le nombre vient d'une variable, il est inconnu à la programmation mais est connu au moment de l'exécution de la boucle
  - le bloc **peut ne pas être exécuté du tout**
  - et une **boucle infinie** est toujours possible
- Contrairement à un WHILE ou DO WHILE, on ne doit pas explicitement modifier la variable de boucle dans le bloc (c'est la boucle qui gère ça elle-même)

BOUCLES IMBRIQUÉES

- Parfois, chaque itération de la boucle nécessite elle-même d'exécuter une boucle
  - ça s'appelle des **boucles imbriquées**

PROBLÈME

TROUVER LA MOYENNE DE CHAQUE ÉTUDIANT D'UNE CLASSE. IL Y A 4 ÉTUDIANTS ET 2 ÉVALUATIONS PAR ÉTUDIANT.

ANALYSE

- On doit boucler sur chaque étudiant pour calculer la moyenne de chacun
  - 4 itérations sur cette boucle
- Mais, pour chaque étudiant, on doit demander 2 notes
  - c'est de nouveau une boucle, à l'intérieur d'une autre boucle ; c'est donc une **boucle imbriquée**
  - 2 itérations sur cette boucle imbriquée

ALGORITHME

- Pour chacun des 4 étudiants
  - Initialiser le total de cet étudiant à 0
  - Pour chacune des 2 évaluations de cet étudiant
    - Demander la note
    - Ajouter la note au total de l'étudiant
  - Calculer la moyenne de l'étudiant (total / 2)
  - Afficher la moyenne

EXEMPLE D'IMPLÉMENTATION

```java
public static void main(String[] args) {
  int nbEtudiants = 4;
  int nbEvals = 2;
  Scanner clavier = new Scanner(System.in);

  for (int numEtudiant = 1; numEtudiant <= nbEtudiants; numEtudiant++) {
    System.out.println("Étudiant " + numEtudiant);
    double total = 0;
    for (int numEval = 1; numEval <= nbEvals; numEval++) {
      System.out.print("Entrez la note de l'évaluation " + numEval + " : ");
      double note = clavier.nextDouble();
      total = total + note;
    }
    double moyenne = total / nbEvals;
    System.out.println("Moyenne de l'étudiant " + numEtudiant + " : " + moyenne);
  }
  clavier.close();
}
```

EXERCICE - JEU DE L'OIE

- Le but est de parcourir les 20 cases du jeu avec 5 lancers de dé
- Un dé à 6 faces sera donc lancé 5 fois
  - si, au bout de 5 lancers, on arrive **exactement** à 20, on gagne
  - si on est au-dessus ou en-dessous, on perd
- La simulation va nécessiter de générer des entiers au hasard
  - utiliser un objet `Random` pour générer des entiers entre 1 et 6 :

```
Random generateur = new Random();
int lancer = generateur.nextInt(6) + 1;
```

JEU DE L'OIE - EXEMPLE DE SORTIE

```
Lancer 1 : vous avez fait 3. Vous êtes sur la case 3 (encore 17 cases)
Lancer 2 : vous avez fait 3. Vous êtes sur la case 6 (encore 14 cases)
Lancer 3 : vous avez fait 6. Vous êtes sur la case 12 (encore 8 cases)
Lancer 4 : vous avez fait 5. Vous êtes sur la case 17 (encore 3 cases)
Lancer 5 : vous avez fait 3. Vous êtes sur la case 20.
Vous avez gagné !
```

JEU DE L'OIE - ÉVOLUTION

- Le jeu doit maintenant recommencer automatiquement autant de fois que nécessaire jusqu'à ce qu'on ait une simulation gagnante
  - on doit voir toutes les parties se dérouler
  - la dernière partie sera donc la seule gagnante
- Le programme doit également afficher le nombre total de simulations qu'il a fallu pour gagner