

# Structures répétitives

## *for, while, do while*

# 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

```
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

```
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 modifiée** 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 avoir 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 (condition);
```

# **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 modifiée** 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

```
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 répétitions (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

```
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 ( $\text{total} / 2$ )
  - Afficher la moyenne

# Exemple d'implémentation

```
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 **toujours** lancé 5 fois
  - si 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 (lancer de dé)
  - vous utiliserez un objet Random pour cela
  - voici le code qui permet de générer un entier entre 1 et 6:

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

# Jeu de l'oie - Exemple de sortie (perdant)

```
Lancer 1 : vous avez fait 4. Vous êtes sur la case 4 (encore 16 cases)
Lancer 2 : vous avez fait 5. Vous êtes sur la case 9 (encore 11 cases)
Lancer 3 : vous avez fait 6. Vous êtes sur la case 15 (encore 5 cases)
Lancer 4 : vous avez fait 5. Vous êtes sur la case 20 (encore 0 cases)
Lancer 5 : vous avez fait 3. Vous êtes sur la case 23 (encore -3 cases)
Vous avez perdu !
```

(notez que, bien que le joueur arrive à 20 après 4 lancers, le 5ème lancer le fait dépasser et il perd la partie)

# Jeu de l'oie - Exemple de sortie (gagnant)

```
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 (encore 0 cases)
Vous avez gagné !
```





## Jeu de l'oie - V2

- Le jeu doit maintenant recommencer automatiquement autant de fois que nécessaire jusqu'à ce qu'on ait une simulation gagnante
  - chaque partie fait toujours 5 lancers de dé
  - 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



# Jeu de l'oie - V3

- On ne joue de nouveau qu'une seule partie (comme la V1)
- Cette fois, on lance le dé jusqu'à ce qu'on arrive pile à 20
- Quand on dépasse 20, on «rebondit» en repartant en arrière, comme dans le vrai jeu. Par exemple:
  - si on est sur la case 18 et qu'on fait 3, on atterrit sur la case 19 (2 pour aller à 20, 1 pour revenir à 19)
  - si maintenant on fait 6, on atterrit sur la case 15 (1 pour aller à 20, 5 pour revenir à 15)
  - si maintenant on fait 5, on atterrit sur la case 20 et le jeu s'arrête
- On ne peut donc pas perdre
- En fin de partie, le programme doit afficher le nombre de lancers de dé qu'il a fallu pour gagner



