# Algèbre de Boole

https://github.com/heig-vd-progim-course/heig-vd-progim2-course

Visualiser le contenu complet sur GitHub à cette adresse.

V. Guidoux, avec l'aide de GitHub Copilot.

# Plus de détails sur GitHub

*Cette présentation est un résumé du contenu complet disponible sur GitHub.*

*Pour plus de détails, consulter le contenu complet sur GitHub ou en cliquant sur l'en-tête de ce document.*

# Objectifs (1/2)

- Lister les opérateurs logiques de base (AND, OR, NOT, XOR).
- Évaluer des expressions booléennes simples et complexes.
- Appliquer les tables de vérité pour valider des expressions logiques.

# Objectifs (2/2)

- Simplifier des expressions booléennes en utilisant les lois de De Morgan.
- Construire des conditions complexes pour contrôler le flux d'un programme.

# Introduction à l'algèbre de Boole

L'algèbre de Boole manipule des valeurs de vérité : **vrai** ( `true` ) ou **faux** ( `false` ).

Nommée d'après George Boole (1815-1864), elle est fondamentale en programmation pour :

- Prendre des décisions ( `if` , `else` )
- Contrôler des boucles ( `while` , `for` )
- Valider des données
- Filtrer des informations

# Le type boolean en Java

Une variable de type `boolean` ne peut prendre que deux valeurs : `true` ou `false`.

```
boolean isConnected = true;
boolean hasError = false;
boolean isValidInput = true;
```

Ce type primitif permet de stocker des états logiques et de construire des expressions conditionnelles.

# Opérateur NOT (!)

L'opérateur `!` inverse une valeur booléenne.

```
boolean isActive = true;
boolean isInactive = !isActive;  // false
```

## Table de vérité

| a | !a |
|---|---|
| true | false |
| false | true |

# Opérateur AND (&&) (1/2)

L'opérateur `&&` retourne `true` seulement si **les deux** opérandes sont `true`.

```
boolean hasPermission = true;
boolean isOwner = false;
boolean isAdmin = true;

boolean canEdit = hasPermission && isOwner; // false

boolean canDelete = hasPermission && isAdmin; // true
```

# Opérateur AND (&&) (2/2)

```
a && b
```

| a | b | a && b |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

# Opérateur OR (||) (1/2)

L'opérateur `||` retourne `true` si **au moins un** des opérandes est `true`.

```
boolean isAdmin = false;
boolean isModerator = true;
boolean isUser = false;

boolean hasAccess = isAdmin || isModerator;  // true

boolean canView = isAdmin || isUser;  // false
```

# Opérateur OR (||) (2/2)

```
a || b
```

| a | b | a \|\| b |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

# Opérateur XOR (^) (1/2)

L'opérateur `^` retourne `true` si **exactement un** des opérandes est `true`.

```
boolean option1 = true;
boolean option2 = true;
boolean option3 = false;

boolean exclusiveChoice = option1 ^ option2;  // false

boolean anotherChoice = option1 ^ option3;  // true
```

# Opérateur XOR (^) (2/2)

```
a ^ b
```

| a | b | a ^ b |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | false |

# Expressions booléennes avec comparaisons

Les opérateurs de comparaison produisent des valeurs booléennes.

```
int age = 25;
boolean isAdult = (age >= 18);  // true

String status = "active";
boolean isActive = status.equals("active");  // true

int score = 85;
boolean hasPassedWithHonors = (score >= 90);  // false
```

# Court-circuit des opérateurs (1/2)

Les opérateurs `&&` et `||` utilisent l'**évaluation court-circuit** : la deuxième expression n'est évaluée que si nécessaire.

**Avec AND (&&)**

```
boolean result = (x != 0) && (10 / x > 2);
```

Si `x == 0`, la première condition est `false`. La division `10 / x` n'est **jamais exécutée**, évitant une erreur de division par zéro.

# Court-circuit des opérateurs (2/2)

## Avec OR (||)

```java
String text = null;
boolean isValid = (text != null) || (text.length() > 0);
```

Si `text == null`, la première condition est `false` mais la deuxième **n'est pas évaluée**, évitant une `NullPointerException`.

**Important** : l'ordre des conditions est crucial pour éviter les erreurs d'exécution.

# Priorité des opérateurs (1/2)

Java définit une priorité entre les opérateurs logiques :

1. `!` (NOT) - priorité la plus élevée
2. `&&` (AND)
3. `||` (OR)
4. `^` (XOR) - priorité la plus basse

# Priorité des opérateurs (2/2)

**Recommandation** : toujours utiliser des parenthèses pour rendre l'ordre d'évaluation explicite.

```
boolean result = a || b && c;  // Ambigu : est-ce (a || b) && c ou a || (b && c) ?
// Réponse: a || (b && c) en raison de la priorité, mais ce n'est pas évident à lire.

boolean result = a || (b && c);  // Clair et lisible
```

# Lois de De Morgan (1/2)

Les lois de De Morgan permettent de transformer des expressions logiques.

**Première loi** : `!(a && b)` est équivalent à `(!a || !b)`.

> *La négation d'une conjonction est équivalente à la disjonction des négations.*

**Exemple** : `!(isAdmin && hasPermission)` est équivalent à `(!isAdmin || !hasPermission)`.

# Lois de De Morgan (2/2)

**Deuxième loi** : `!(a || b)` est équivalent à `(!a && !b)`.

> *La négation d'une disjonction est équivalente à la conjonction des négations.*

**Exemple** : `!(isWeekend || isHoliday)` est équivalent à `(!isWeekend && !isHoliday)`.

Ces lois sont utiles pour simplifier ou restructurer des conditions complexes.

# Applications pratiques : structures conditionnelles

```java
boolean hasLocalFiles = true;
boolean hasInternetConnection = false;

if (hasLocalFiles && !hasInternetConnection) {
    System.out.println("Mode hors ligne activé.");
} else if (hasLocalFiles && hasInternetConnection) {
    System.out.println("Synchronisation des fichiers...");
} else {
    System.out.println("Aucun fichier disponible.");
}
```

# Applications pratiques : boucles

Les conditions booléennes contrôlent quand une boucle continue ou s'arrête.

```
int attempts = 0;
boolean isSuccess = false;
int maxAttempts = 3;

while ((attempts < maxAttempts) && !isSuccess) {
    System.out.println("Tentative " + (attempts + 1));
    isSuccess = tryConnect();
    attempts++;
}
```

# Applications pratiques : conditions complexes

Combiner plusieurs critères pour des décisions sophistiquées.

```
boolean canUseService = (age >= 18) &&
                        hasValidID &&
                        (hasSubscription || hasTrialPeriod);

if (canUseService) {
    System.out.println("Accès autorisé au service.");
} else {
    System.out.println("Accès refusé.");
}
```

# Bonnes pratiques : utiliser les parenthèses

Toujours utiliser des parenthèses pour rendre l'ordre d'évaluation explicite.

### À éviter

```
boolean result = a || b && c;  // Ambigu
```

### À privilégier

```
boolean result = a || (b && c);  // Clair
```

Le code est lu plus souvent qu'il n'est écrit.

# Bonnes pratiques : noms de variables explicites

Utiliser des préfixes comme `is`, `has`, `can` pour les variables booléennes.

## À éviter

```
boolean x = true;
boolean flag = false;
```

## À privilégier

```
boolean isConnected = true;
boolean hasPermission = false;
boolean canEdit = (age > 18);
```

# Bonnes pratiques : éviter les comparaisons redondantes

Pas besoin de comparer une variable booléenne à `true` ou `false`.

## À éviter

```
if (isReady == true) { }
if (hasError == false) { }

return isValid == true;
```

## À privilégier

```
if (isReady) { }
if (!hasError) { }

return isValid;
```

# Bonnes pratiques : décomposer les expressions complexes

```
// Complexe et difficile à lire
if ((user.isActive() && user.hasSubscription() && !user.isBanned()) ||
    (user.isAdmin() && user.hasValidToken())) {
    // ...
}

// Plus clair et maintenable
boolean isRegularUser = user.isActive() && user.hasSubscription() && !user.isBanned();
boolean isAuthenticatedAdmin = user.isAdmin() && user.hasValidToken();

if (isRegularUser || isAuthenticatedAdmin) {
    // ...
}
```

# Erreurs courantes : confondre = et ==

`=` est l'affectation, `==` est la comparaison.

## Erreur

```
if (isActive = false) {  // Affectation !
    System.out.println("Inactif");
}
```

## Correction

```
if (!isActive) {  // Comparaison
    System.out.println("Inactif");
}
```

# Erreurs courantes : types non booléens

En Java, les opérateurs logiques ne fonctionnent qu'avec des expressions booléennes.

## Invalide en Java

```
int count = 0;
if (count) {  // ERREUR en Java
    // ...
}
```

## Correction

```
if (count != 0) {  // Expression booléenne explicite
    // ...
}
```

# Erreurs courantes : oubli de l'ordre avec court-circuit

```
// Risque de NullPointerException si text est null
String text = null;
if ((text.length() > 0) && (text != null)) {  // NullPointerException !
    // ...
}

// Correct : vérifier d'abord que text n'est pas null
if ((text != null) && (text.length() > 0)) {  // Sûr
    // ...
}
```

# Questions

Est-ce que vous avez des questions ?

# À vous de jouer !

- (Re)lire le contenu de cours.
- Lire les exemples de code et les descriptions.
- Faire les exercices.
- Faire le mini-projet.
- Poser des questions si nécessaire.
- Entraidez-vous

➡️ Contenu complet sur GitHub.

# Sources

- Illustration principale par Shubham Dhage sur Unsplash
- Illustration par Aline de Nadai sur Unsplash
- Illustration par Desola Lanre-Ologun sur Unsplash
- Illustration par Nikita Kachanovsky sur Unsplash