Boolean Algebra & Logic

What Problem This Solves

Boolean algebra is the mathematics of true/false decisions.

Every time you write:

- `if (user.isActive && user.hasPermission)`
- `WHERE status IN ('active', 'pending') AND created_at > '2024-01-01'`
- Feature flags, permission checks, filter logic
- Circuit design, search queries, validation rules

...you're using Boolean algebra.

It's the foundation of all computational logic.

Intuition & Mental Model

Think: Light Switches and Gates

Boolean: Only two values

```
true / false
1 / 0
on / off
yes / no
```

Boolean operations: Combining switches

```
Series (AND):        Parallel (OR):
  ┌──S1──S2──●         ┌──S1──┐
  └────────────┘       ├──S2──┼──●
                       └──────┘


Both must be on      Either can be on
```

Core Concepts

1. Basic Operations

AND (∧)

```
true AND true = true
true AND false = false
false AND true = false
false AND false = false

Only true if BOTH are true
```

```javascript
const canAccess = user.isLoggedIn && user.hasPermission;

// Short-circuit: if first is false, doesn't check second
if (user && user.profile) { ... }
```

OR (∨)

```
true OR true = true
true OR false = true
false OR true = true
false OR false = false

True if AT LEAST ONE is true
```

```javascript
const shouldNotify = user.emailEnabled || user.smsEnabled;

// Short-circuit: if first is true, doesn't check second
if (isAdmin || hasSpecialAccess) { ... }
```

NOT (¬)

```
NOT true = false
NOT false = true

Flips the value
```

```javascript
const isInactive = !user.isActive;

// Double negative
const isActive = !!user.isActive; // Converts to boolean
```

---

2. Truth Tables

AND

A	B	A AND B
T	T	**T**
T	F	F
F	T	F
F	F	F

OR

A	B	A OR B
T	T	**T**

T	F	**T**
F	T	**T**
F	F	F

NOT

A	NOT A
T	F
F	T

XOR (Exclusive OR)

A	B	A XOR B
T	T	F
T	F	**T**
F	T	**T**
F	F	F

```
True if EXACTLY ONE is true (not both)
```

```
// XOR in JavaScript
const xor = (a, b) => (a || b) && !(a && b);

// Or using bitwise
const xorBit = (a, b) => a ^ b;

// Example: toggle feature
let darkMode = false;
darkMode = !darkMode; // XOR with true
```

---

3. De Morgan's Laws (Critical)

Law 1: NOT(A AND B) = (NOT A) OR (NOT B)

```
// These are equivalent:
if (!(user.isActive && user.hasPermission)) { ... }
if (!user.isActive || !user.hasPermission) { ... }
```

Law 2: NOT(A OR B) = (NOT A) AND (NOT B)

```
// These are equivalent:
if (!(isPremium || isBeta)) { ... }
```

```
if (!isPremium && !isBeta) { ... }
```

Why it matters:

```
// Readable version
if (user.isActive && user.hasPermission) {
  grantAccess();
}

// Equivalent (early return pattern)
if (!user.isActive || !user.hasPermission) {
  return;
}
grantAccess();
```

Visual:

```
NOT(A AND B):
NOT(●──●) = (○──○) | (○──○)

If not both on = If either is off
```

---

4. Operator Precedence

Order of operations (like PEMDAS for logic):

```
1. NOT (¬)
2. AND (∧)
3. OR (∨)
```

Example:

```
// What is the value?
true || false && false

// Step 1: AND first
false && false = false

// Step 2: OR
true || false = true

// Result: true
```

Use parentheses for clarity:

```
// Ambiguous
if (isAdmin || isPremium && hasAccess) { ... }
```

```
// Clear
if (isAdmin || (isPremium && hasAccess)) { ... }
```

---

5. Boolean Expressions

Compound conditions:

```
// Complex access control
const canEdit = (
  user.isAuthor ||
  user.isAdmin ||
  (user.isEditor && document.isPublished)
);
```

Simplification (using algebra):

```
// Before
if ((A && B) || (A && C)) { ... }

// After (factor out A)
if (A && (B || C)) { ... }

// Fewer checks, same result
```

Common patterns:

```
// Guard clauses (early exit)
if (!user) return;
if (!user.isActive) return;
if (!user.hasPermission) return;
doSomething();

// Equivalent to
if (user && user.isActive && user.hasPermission) {
  doSomething();
}
```

---

6. Bitwise Operations

Boolean algebra on bits:

```
const A = 0b1010; // 10
const B = 0b1100; // 12

// AND
A & B;  // 0b1000 = 8

// OR
A | B;  // 0b1110 = 14
```

```
// XOR
A ^ B;  // 0b0110 = 6

// NOT
~A;     // 0b...11110101 (inverts all bits)
```

Common use cases:

```
// Permissions (bitmask)
const READ = 0b001;   // 1
const WRITE = 0b010;  // 2
const EXECUTE = 0b100; // 4

const permissions = READ | WRITE; // 0b011 = 3

// Check permission
const canRead = (permissions & READ) !== 0; // true
const canExecute = (permissions & EXECUTE) !== 0; // false

// Add permission
permissions |= EXECUTE; // Now 0b111 = 7

// Remove permission
permissions &= ~WRITE; // Now 0b101 = 5
```

Software Engineering Connections

1. Conditional Logic

```
// User access control
function canAccessResource(user, resource) {
  return (
    user.isAdmin ||
    (user.id === resource.ownerId) ||
    (user.groups.some(g => resource.allowedGroups.includes(g)))
  );
}

// Short-circuit optimization
if (user.isAdmin) return true; // No need to check further
```

2. Database Queries

```
-- Boolean algebra in WHERE clauses
SELECT * FROM users
WHERE (status = 'active' AND email_verified = true)
   OR (status = 'pending' AND created_at > NOW() - INTERVAL '7 days');
```

```sql
-- De Morgan's law
SELECT * FROM users
WHERE NOT (status = 'inactive' OR email_verified = false);

-- Equivalent to:
SELECT * FROM users
WHERE status != 'inactive' AND email_verified = true;
```

3. Feature Flags

```javascript
const features = {
  darkMode: true,
  betaFeatures: false,
  analytics: true
};

// Enable feature if any flag is on
const showBanner = features.darkMode || features.betaFeatures;

// Enable only if all conditions met
const enableAdvanced = features.betaFeatures && user.isPremium;
```

4. Form Validation

```javascript
function validateForm(form) {
  const isValid = (
    form.email &&
    form.email.includes('@') &&
    form.password &&
    form.password.length >= 8 &&
    form.terms Accepted
  );

  return isValid;
}

// Early validation (fail fast)
if (!form.email) return false;
if (!form.email.includes('@')) return false;
if (!form.password) return false;
// ... etc
```

5. React Conditional Rendering

```jsx
// AND operator for conditional render
{user && user.isAdmin && <AdminPanel />}
```

```
// OR for fallback
{user.avatar || <DefaultAvatar />}

// Complex conditions
{(isPremium || isTrial) && !hasExpired && (
  <PremiumFeature />
)}
```

6. API Filters

```
// Query builder
const query = {};

if (filters.status) {
  query.status = filters.status;
}

if (filters.minDate || filters.maxDate) {
  query.created_at = {};
  if (filters.minDate) query.created_at.$gte = filters.minDate;
  if (filters.maxDate) query.created_at.$lte = filters.maxDate;
}

// Boolean combination
if (filters.isActive && filters.isVerified) {
  query.$and = [
    { is_active: true },
    { is_verified: true }
  ];
}
```

Common Misconceptions

❌ "AND means 'also check this'"

AND means 'both must be true', not "add another condition". Order doesn't matter in logic (but can affect performance).

```
// Same result
A && B === B && A

// But different performance
if (expensiveCheck() && cheapCheck()) { ... } // Bad
if (cheapCheck() && expensiveCheck()) { ... } // Good (short-circuit)
```

❌ "OR means 'pick one'"

OR means 'at least one', not "exactly one" (that's XOR).

```
// OR: true if either or both
true || true === true

// XOR: true if exactly one
true ^ true === false
```

❌ "!= is the same as NOT"

Different operators:

```
// NOT (logical)
!true === false

// != (comparison)
5 != 3 === true
5 != 5 === false
```

❌ "Double negatives are redundant"

Sometimes useful for **type coercion**:

```
const value = "hello";
!!value; // true (converts to boolean)

// Equivalent to
Boolean(value); // true
```

❌ "Bitwise and logical are the same"

Different purposes:

```
// Logical (short-circuit)
true && false; // false (doesn't eval second if first is false)

// Bitwise (always evaluates both)
1 & 0; // 0 (operates on bits)
```

Practical Mini-Exercises

Exercise 1: Simplify Logic

Simplify this condition:

```
if ((isAdmin || isModerator) && (isAdmin || isActive)) {
  grantAccess();
}
```

▶ Solution

Exercise 2: De Morgan's Law

Rewrite using De Morgan's laws:

```
if (!(user.isPremium && user.isVerified)) {
  showUpgradePrompt();
}
```

▶ Solution

Exercise 3: Permission Bitmask

Implement a permission system using bitwise operations:

```
const PERMISSIONS = {
  READ: 1,    // 0b001
  WRITE: 2,   // 0b010
  DELETE: 4   // 0b100
};

// User has READ and WRITE
let userPerms = PERMISSIONS.READ | PERMISSIONS.WRITE;

// Check if user has DELETE
// Add WRITE permission
// Remove READ permission
```

▶ Solution

Summary Cheat Sheet

Basic Operations

Operation	Symbol	JavaScript	Example
AND	∧	`&&`	`true && false` → false
OR	∨	`||`	`true || false` → true
NOT	¬	`!`	`!true` → false
XOR	⊕	`^`	`true ^ true` → false

De Morgan's Laws

```
// Law 1
!(A && B) === (!A || !B)

// Law 2
!(A || B) === (!A && !B)
```

Precedence

```
NOT > AND > OR

Example: A || B && C
= A || (B && C)
```

Bitwise Operations

```
A & B    // AND
A | B    // OR
A ^ B    // XOR
~A       // NOT
```

Common Patterns

```
// Early exit (guard clause)
if (!condition) return;

// Default value
const value = userInput || defaultValue;

// Conditional assignment
const status = isActive ? 'active' : 'inactive';

// Permission check
const canEdit = isOwner || isAdmin;
```

Next Steps

Boolean algebra is the foundation of all logic in software. You now understand how to combine conditions, simplify expressions, and work with permissions.

Next, we'll explore **asymptotic thinking**—understanding how algorithms scale and why Big-O notation matters.

Continue to: 04-asymptotic-thinking.md