L04

AI Game Development & Computer Science Summer Program

Lecture 4: Control Flow and Boolean Logic

From functions to predicates

A predicate is a function that answers yes or no.

```
const hasEnoughEnergy = (energy:
number): boolean => {
  return energy >= 10;
};

hasEnoughEnergy(12); // true
```

Function contract

- Input: one number.
- Output: one boolean.
- Question: is the energy high enough?

Naming habit

- Start predicates with is, has, or can.
  The name should read like a yes-or-no question.

Booleans have only two values

true and false are values, not strings.

Boolean

- `true`
- `false`
- Useful for decisions.

Comparison

- `score >= 70`
- `name === "Ada"`
- `energy < cost`

Result

- Every comparison evaluates to a boolean.
- Trace it before using it in an `if`.

Use `===` for exact equality. It keeps value and type checks honest.

Boolean operators combine questions

&&, ||, and ! build bigger boolean expressions.

Operator	Reads as	True when
a && b	a and b	both are true
a || b	a or b	at least one is true
!a	not a	a is false

Trace rule

- Evaluate the smaller comparisons first.
- Then combine their boolean results.
- Add parentheses when the idea could be misread.

Truth tables make every case visible

Use a table when a decision depends on multiple booleans.

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

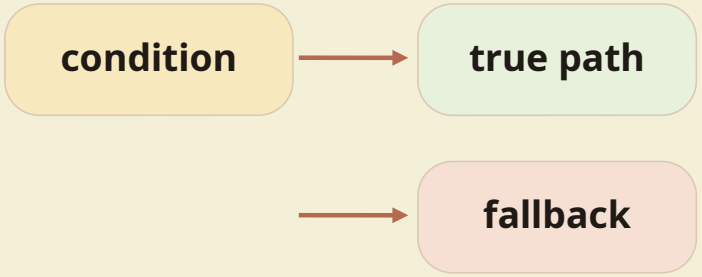
Why this helps

- No guessing.
- No missing edge case.
- You can compare the table to the code line by line.

A conditional chooses one path

An `if` statement runs code only when its condition is true.

```
const actionForEnergy = (energy: number):
string => {
  if (energy >= 10) {
    return "attack";
  }
  return "rest";
};
```



Trace it

- For `energy = 12`, the first return runs.
- For `energy = 4`, the fallback return runs.

Branch order matters

The first matching branch wins.

```
const rankScore = (score: number): string => {
  if (score >= 70) return "review";
  if (score >= 90) return "strong";
  return "retry";
};

rankScore(95); // What happens?
```

Bug

- A score of 95 matches `score >= 70` first.
- The `strong` branch is never reached.

Fix

- Put the most specific condition first.
- Then move toward the broader fallback.

Guard clauses handle edge cases early

A guard exits before the main rule runs.

```
const spendEnergy = (energy: number, cost:
number): number => {
  if (cost < 0) return energy;
  if (energy < cost) return energy;
  return energy - cost;
};
```

Why guards help

- They make invalid or special cases visible.
- They keep the main rule short.
- They reduce nested `if` statements.

Read guards top to bottom: each one answers, should we stop here?

Combine conditions into a decision

Write the decision in plain language before code.

```
const canUsePowerUp = (
  hasItem: boolean,
  energy: number,
  isPaused: boolean,
): boolean => {
  return hasItem && energy >= 10 && !isPaused;
};
```

Plain language

- You can use the power-up when:
- you have the item,
- you have enough energy,
- and the game is not paused.

Be explicit instead of relying on truthiness

Some values act false-like, but clarity wins early.

Value	Boolean context	Clear check
0	false-like	score === 0
""	false-like	name.length === 0
[]	truthy	items.length === 0

Course habit

- Write the comparison you mean.
- Do not make the reader remember language tricks.
- Let TypeScript narrow from explicit checks.

AI can help with decision tables

Ask for missing cases, then verify the logic yourself.

Prompt

- Here are three boolean inputs and the intended rule.
- Create a truth table with expected outputs.
- Flag any contradictory or unreachable cases.

Your review

- Check each row against the plain-language rule.
- Run the function on boundary values.
- Reject output that changes the contract.

AI can make the table wider. You still own the rule.

Studio build: choose the next action

Turn a decision table into a typed helper

```
type Action = "attack" | "rest" | "shield";

const chooseAction = (
  energy: number,
  enemyClose: boolean,
  shieldReady: boolean,
): Action => {
  // Fill in the guards and branches.
};
```

Build order

- Write the plain-language rule.
- Make a small decision table.
- Implement the branches.
- Test normal and boundary cases.
- Commit locally if the checkpoint is ready.

Constraint

- Do not add a branch unless the table needs it.

Debugging exercise: can this ever happen?

Unreachable branches usually mean the conditions are out of order or contradictory.

```
const badge = (score: number): string => {
  if (score >= 50) return "bronze";
  if (score >= 75) return "silver";
  if (score >= 90) return "gold";
  return "try again";
};
```

Find it

- Which branches are unreachable?
- What input proves the bug?
- How should the branches be reordered?

Check

- A truth table or sorted examples can prove the fix.

Lab plan: practice, build, apply

Use the main lab block to turn the concept into working evidence.

Guided practice

- Convert a decision table into conditions.
- Trace true and false branches separately.
- Review `git status` before starting the studio checkpoint.

Build target

- Create a next-action helper from a game state.
- Test normal, edge, and impossible-looking cases.

Project time

- Add one P1 decision rule.
- Write the table before the function.
- Update the checkpoint note before the local Git checkpoint.
- Create the first local Git checkpoint if your rule and test are ready.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Refactor nested conditions into guard clauses.
- Add a tie or priority rule.
- Ask AI for missing branches, then verify each one.

Done when

- The main lab still works.
- You can explain the tradeoff or edge case.
- You saved one proof object: test, trace, screenshot, or note.

P1: Game Rules Milestone

L03-L15

Checkpoint

Move on when you can explain the condition before reading the branch body.

Exit ticket

- One predicate name that reads like a question.
- One boolean expression traced step by step.
- One guard clause for an edge case.
- One local checkpoint message or changed-file note.

Next class preview

- Scope.
- State.
- Iteration.
- Repeated decisions over changing values.