L05

AI Game Development & Computer Science Summer Program

Lecture 5: Scope, State, and Iteration

A variable name points to a value

Reassignment changes the name's binding, not the old value.

```
let score = 10;
score = 20;

// 10 did not change.
// score now points to 20.
```

Use const for stable names

- The name cannot be reassigned.
- The code becomes easier to trust.

Use let for changing loop state

- Counters.
- Accumulators.
- Values that intentionally update over time.

Scope is where a name exists

A name is only usable inside the region where it was defined.

Outer scope

- Names defined outside a function or block can be seen by inner code.
- Keep this small so surprises stay small.

Function scope

- Parameters and helper variables live inside the function.
- They disappear after the call finishes.

Block scope

- Names declared inside `{}` stay inside that block.
- This is why loop variables do not leak out.

Debugging habit: ask, where was this name created?

Trace local variables inside a function

Each function call gets its own local names.

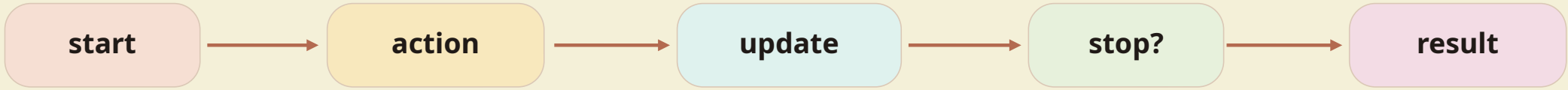
```
const addBonus = (score: number): number => {
  const bonus = 5;
  const total = score + bonus;
  return total;
};

addBonus(10);
```

Name	Value	Where it exists
score	10	inside this call
bonus	5	inside this call
total	15	inside this call

After the function returns, the result remains. The local names are gone.

Iteration means repeat with control

Every loop needs a start, a repeated action, an update, and a stop.



Loop control

- A loop is correct only when it stops at the right time.
- A loop bug is often a missing update or a wrong stop condition.

for...of loops over a list

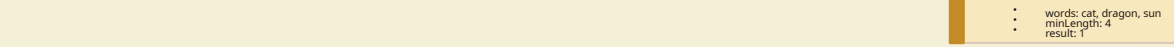
Use this when you want each value in a collection.

```
const countLongWords = (words: string[], minLength: number):
number => {
  let count = 0;
  for (const word of words) {
    if (word.length >= minLength) count += 1;
  }
  return count;
};
```

What changes

- `word` changes to the next list item each time.
- `count` changes only when the condition is true.
- The list itself does not need to change.

Trace target



Trace the loop state

A trace table makes hidden updates visible.

word	length >= 4?	count after
cat	false	0
dragon	true	1
sun	false	1

What to notice

- The loop variable changes each row.
- The accumulator keeps its value between rows.
- The returned value is the final accumulator.

A counted loop gives you an index

Use an index when position matters.

```
const sumSquaresTo = (limit: number): number => {
  let total = 0;
  for (let i = 1; i <= limit; i += 1) {
    total += i * i;
  }
  return total;
};
```

i	i * i	total
1	1	1
2	4	5
3	9	14

The index exists inside the loop. The total survives because it was defined before the loop.

while loops run until a condition changes

Use this when the number of repeats is not known first

```
const turnsUntilEmpty = (energy: number, cost: number):
number => {
  let turns = 0;
  while (energy >= cost) {
    energy -= cost;
    turns += 1;
  }
  return turns;
};
```

Stop condition

- The loop continues while `energy >= cost`.
- Inside the loop, energy must move toward stopping.
- If the condition never changes, the loop can run forever.

State is memory that changes over time

Loop state should be small, named, and intentional.

Loop state

- Short-lived.
- Used to compute one result.
- Example: `count`, `total`, `i`.

Program state

- Longer-lived.
- Represents something the program remembers.
- Example: score, inventory, current level.

Habit

- Keep loop state inside the helper when possible.
- Return the result instead of changing distant names.

Common loop bugs

Most loop bugs are start, update, or stop mistakes.

Off by one

- The loop runs one too many or one too few times.
- Check `<` versus `<=`.

Missing update

- The condition never changes.
- The loop might never stop.

Wrong scope

- A name is created inside a block but used outside it.
- Move the name or return the value.

When a loop looks wrong, trace three rows before editing.

AI can help make trace tables

Ask for examples. Keep control of the reasoning.

Prompt

- Here is a TypeScript loop.
- Make a trace table for three inputs.
- Include start value, each update, and final result.

Your review

- Check the first row by hand.
- Check the stopping condition.
- Add one edge case the AI missed.

Generated traces are drafts until you can explain each update.

Studio build: count matching words

Use scope and state to process repeated structure.

```
const countMatches = (words: string[], target: string):
number => {
  let count = 0;
  for (const word of words) {
    if (word === target) count += 1;
  }
  return count;
};
```

Build order

- Say the domain and range.
- Trace three words in a table.
- Test zero matches, one match, and repeated matches.

Extension

- Change exact match into minimum length or starts-with logic.

Lab plan: practice, build, apply

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

Guided practice

- Trace which names exist inside and outside a block.
- Walk through a loop update step by step.
- Spot one accidental shared state bug.

Build target

- Count matching words or events with a loop.
- Explain each update to the accumulator.

Project time

- Add repeated turns or attempts to P1.
- Decide what state changes after each turn.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Add a reset rule.
- Trace the loop with an empty list.
- Ask AI to produce a buggy loop and fix it.

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 point to each name's lifetime and each loop update.

Exit ticket

- One name that is local to a function.
- One loop variable and one accumulator.
- One stop condition that prevents an infinite loop.
- One AI-generated trace row you corrected or verified.

Next class preview

- ListNode chains.
- Data modeling.
- Using linked collections as structured program data.