L07

AI Game Development & Computer Science Summer Program

Lecture 7: Higher-Order Functions

A higher-order function uses another function

A function can be passed around like other values.

Regular function

- Takes values.
- Returns a value.
- Example: `rankScore(score)`.

Higher-order function

- Takes a function, returns a function, or both.
- Example: `countWhere(first, isOpen)`.

Why it helps

- You name the repeated pattern once.
- Then you supply the rule that changes.

A callback is the rule you hand in

The traversal helper controls the walk. Your callback controls one-node logic.

```
type Task = { title: string; done: boolean };
type ListNode = { value: Task; next?: ListNode };

const isOpen = (task: Task): boolean => !task.done;

countWhere(firstTask, isOpen);
```

Callback contract

- Input: one node value.
- Output: the answer for that value.
- TypeScript checks both sides.

You still trace the chain one node at a time. The callback only names the rule.

A transform rule visits every value

The trace is value in, callback, value out.

```
const labelTask = (task: Task): string => {
  return task.title.toUpperCase();
};

// The traversal applies labelTask to each node
value.
```

Input	Callback	Output
trace loop	labelTask	TRACE LOOP
fix bug	labelTask	FIX BUG
test edge	labelTask	TEST EDGE

A keep rule decides whether a value counts

```
const countWhere = (
  node: ListNode | undefined,
  keep: (task: Task) => boolean,
): number => {
  let count = 0;
  let current = node;
  while (current) {
    if (keep(current.value)) count += 1;
    current = current.next;
  }
  return count;
};
```

Callback contract

- Input: one task.
- Output: boolean.
- true means count it.
- false means move on.

The helper does the walking. The callback only answers one value at a time.

Optional extension: combine reduces many values into one

U

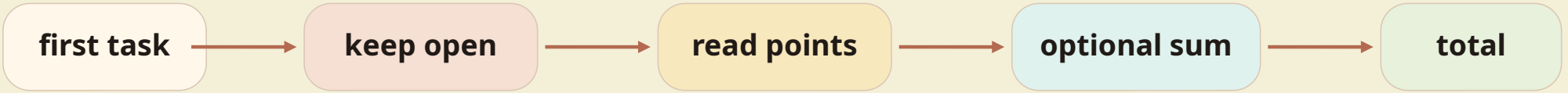
```
const sumPoints = (node: ListNode | undefined):
number => {
  let total = 0;
  let current = node;
  while (current) {
    total = total + current.value.points;
    current = current.next;
  }
  return total;
};
```

total before	points	total after
0	10	10
10	20	30
30	5	35

The starting value matters. It is the first accumulator state.

Helper order changes meaning

Read the helper sequence left to right and say what each step produces.



Trace question

- After each stage, what type do we have?
- After each stage, how many values do we have?
- Does the next stage expect that shape?

Loop or callback helper?

Choose the form that makes the idea easier to read.

Plain loop is often clearer when

- You need several changing states.
- You need early exit.
- The rule has many branches.

Callback helper is often clearer when

- The traversal pattern repeats.
- The callback has one job.
- The helper name explains the intent.

Higher-order functions are tools for clarity, not a contest to hide every loop.

TypeScript checks callback contracts

The callback input comes from the node value type.

```
type Task = { title: string; points: number };

type Predicate = (task: Task) => boolean;
const highValue: Predicate = (task) =>
task.points >= 10;
```

What to watch

- The callback receives a value, not the whole chain.
- The helper decides when to follow `next`.
- Combine-style helpers may return a different type.

Debugging habit

- Hover or inspect the callback parameter type.

AI can review a callback helper

Ask AI to explain the callback contract, then verify the trace.

Prompt

- Here is a TypeScript node-chain helper.
- List the callback input type, output type, and stop behavior.
- Suggest one edge case.

Your review

- Check the callback contract.
- Trace one tiny chain by hand.
- Reject rewrites that skip nodes or hide traversal.

Generated explanations are useful only after you can trace one item yourself.

Studio build: count matching nodes

```
type Task = { title: string; points: number; done: boolean };
type ListNode = { value: Task; next?: ListNode };

const isDone = (task: Task): boolean => task.done;

const doneCount = countWhere(firstTask, isDone);
```

Build order

- Trace the plain traversal.
- Name the callback contract.
- Run the helper on empty, one-node, and three-node chains.
- Optional: add combine after this works.

Lab plan: practice, build, apply

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

Guided practice

- Trace transform and keep/drop callbacks before using them.
- Name each callback by input and output.
- Compare a plain traversal and a callback helper for readability.

Build target

- Rewrite a score or inventory traversal with a predicate callback.
- Keep each callback small enough to test.

Project time

- Use a callback helper in one P1 helper.
- Explain why the helper is readable.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Add a combine helper after counting works.
- Use reduce-style accumulation for one summary value and trace the accumulator.
- Ask AI for a shorter version and check if it is clearer.

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 each helper step has a clear input and output.

Exit ticket

- One callback and its input type.
- One transform trace.
- One keep predicate.
- One place a plain traversal is clearer.
- One AI suggestion you accepted or rejected.

Next class preview

- Events and interactive programs.
- Functions that run later.
- Keeping state changes deliberate.