L03

AI Game Development & Computer Science Summer Program

Lecture 3: Functions as Abstractions

A function is a named rule

Give it inputs. It produces one output.

```
plus5(3) -> 8
```

What the name does

- Hides the details behind a useful idea.
- Lets you reuse the rule without rewriting it.
- Makes the program easier to explain.

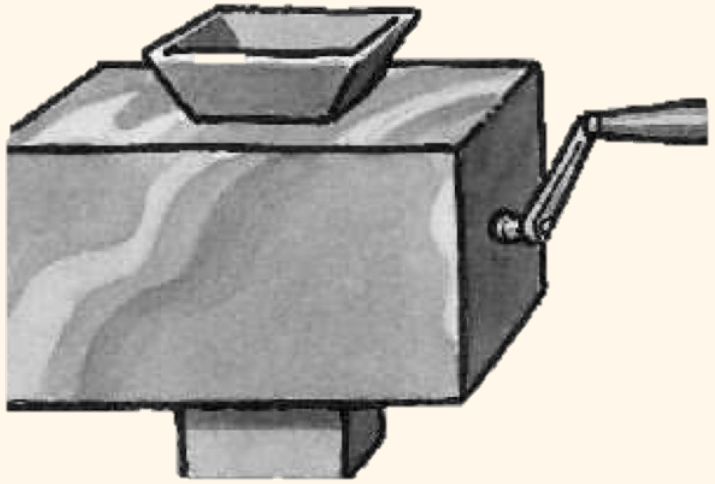
What the call does

- Supplies concrete input values.
- Runs the rule one time.
- Gives back one result.

A good function name says what question the code answers.

The function-machine model

Inputs go in. A rule runs. One output comes out.



Source visual: function machine

Use this model while tracing

- Circle the function name.
- Label each input.
- Write the output before moving on.

Important constraint

For the same input, a pure function gives the same output.
That makes it easier to test and trust.

A TypeScript function has a contract

The contract tells us what goes in and what comes out.

```
const plus5 = (n: number): number =>
{
  return n + 5;
};
```

Input contract

- `n: number` means the input must be a number.
- The function body can rely on that promise.

Output contract

- `: number` means the return value must be a number.
- TypeScript checks the promise for us.

Parameters, arguments, and returns

These words keep the conversation precise.

Parameter

- The input name in the function definition.
- Example: `score` in `scoreRank(score)`.

Argument

- The actual value supplied during a call.
- Example: `87` in `scoreRank(87)`.

Return value

- The single result the function gives back.
- Example: `"strong"`.

When debugging, name which part is wrong: the parameter, the argument, or the returned value.

Domain and range make contracts clearer

Types are useful, but they are not the whole contract.

Domain

- Allowed inputs.
- Type: `number`.
- Extra rule: score is from 0 to 100.

Function

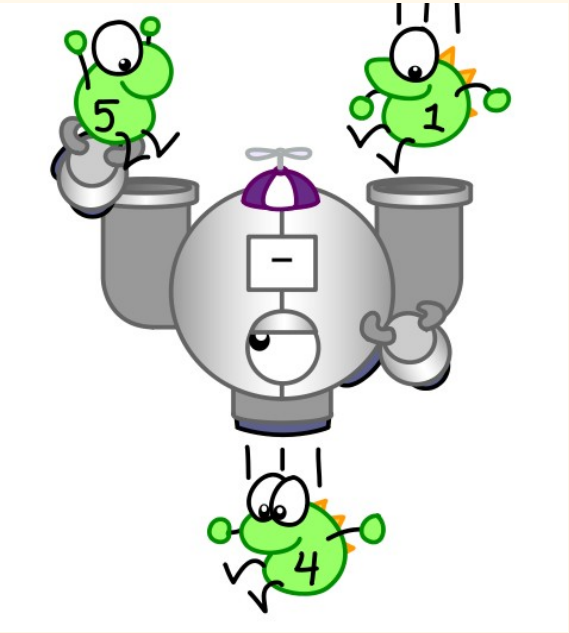
- `scoreRank(score)`
- The rule maps an allowed score to a label.

Range

- Possible outputs.
- Type: `"retry" | "review" | "strong"`.
- Only these labels should appear.

Pure functions are easier to test

A pure function depends only on its inputs.



Source visual: input-to-output idea

Pure function

- Same input gives same output.
- No saved value is changed.
- Nothing extra happens while it runs.

Side effect

- Printing text.
- Writing to a file.
- Changing a saved value.

Side effects are not evil. Keep them intentional and separate from the rule you want to test.

Trace one function call

Do the substitution step before you trust the result.

```
const damage = (power: number, armor: number):
number => {
  return Math.max(0, power - armor);
};

const hit = damage(12, 5);
```

Trace

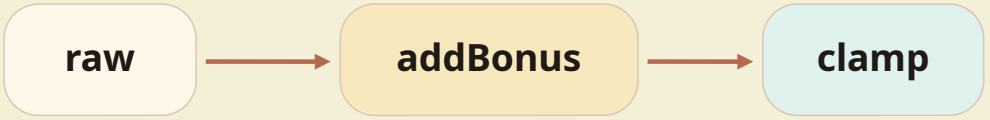
- power becomes 12.
- armor becomes 5.
- Math.max(0, 12 - 5) becomes 7.
- hit stores 7.

If you can trace it, you can debug it.

Composition: plug functions together

A bigger result can come from small functions with clear contracts.

```
const clamp = (n: number): number => Math.max(0,
n);
const addBonus = (n: number): number => n + 10;
const finalScore = (raw: number): number =>
clamp(addBonus(raw));
```



Good composition habit

- Each function does one job.
- The output type of one fits the input type of the next.
- You can test each piece separately.

Decomposition: break a problem into functions

When a problem feels large, name the smaller questions.

Step 1

- What raw values do we have?

Step 2

- What result do we need?

Step 3

- Which small rules connect them?

Step 4

- Which edge cases can break a rule?

Example

`finalScore` can be split into `addBonus`, `clamp`, and `rankScore`.
Small names make the big idea easier to read.

Test the contract, not just the happy path

A good test asks whether the rule handles the edges.

```
type Rank = "retry" | "review" | "strong";

const rankScore = (score: number): Rank =>
  score >= 90 ? "strong" : score >= 70 ?
"review" : "retry";
```

Cases to test

- A normal low score: 65.
- A boundary score: 70.
- A boundary score: 90.
- A suspicious score: -5 or 105.

The suspicious cases reveal whether the domain rule is written down or only assumed.

AI can help generate test ideas

Use AI to widen your checklist. You still decide what is correct.

Prompt

- Here is a TypeScript function and its intended domain.
- List normal, boundary, and invalid test cases.
- Do not rewrite the function yet.

Your review

- Check whether every proposed input is allowed.
- Predict the expected output yourself.
- Reject tests that do not match the contract.

AI suggestions are useful only after you can explain the function contract.

Studio build: game-scoring helpers

Build tiny functions before combining them.

```
type Rank = "retry" | "review" | "strong";

const addComboBonus = (score: number, combo: number): number =>
score + combo * 5;
const capScore = (score: number): number => Math.min(100,
Math.max(0, score));
const rankScore = (score: number): Rank => score >= 90 ?
"strong" : score >= 70 ? "review" : "retry";
```

Build order

- Run `git status` before editing.
- Trace each helper with one input.
- Add one boundary test.
- Compose the helpers into `finalRank`.

Target

- A player score turns into one clear rank.
- Every helper has a contract you can say aloud.

Debugging exercise: find the contract bug

The code runs, but the rule is incomplete.

```
const capScore = (score: number): number
=> {
  return Math.min(100, score);
};

capScore(-20); // What should happen?
```

Questions

- What is the intended domain?
- What is the current range?
- Which boundary case exposes the bug?
- What one-line change fixes it?

A bug is often a missing sentence in the contract.

Lab plan: practice, build, apply

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

Guided practice

- Turn three rules into function signatures.
- Trace one function with two different inputs.
- Run `git status` before editing so you know the starting point.

Build target

- Build one game-scoring helper with a clear return type.
- Write one normal-case test and one edge-case test using the template.

Project time

- Start P1 with one working pure helper, then add a second helper if time allows.
- Write the contract before the code.
- Fill a checkpoint note with what changed, command evidence, and the next step.
- Run `git status` again and explain which files changed.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Compose two helpers into one larger rule.
- Add a boundary case for a maximum or minimum score.
- Ask AI to critique the function names.

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 the function's contract is visible in your words and tests.

Exit ticket

- One function name that hides useful detail.
- One parameter, one argument, and one return value.
- One domain rule that a type alone cannot express.
- One changed file from `git status` that you can explain.

Next class preview

- Boolean logic.
- Conditional branches.
- Functions that choose between outcomes.