L01

AI Game Development & Computer Science Summer Program

Lecture 1: Computing, Abstraction, and Programming

Today's path

One setup check, one durable computing idea, one tiny interaction plan.

1. **Setup check**
   Green, yellow, or red tells us what help you need.
2. **Computing is modeling**
   We choose what matters and leave the rest out.
3. **Abstraction has two moves**
   Remove detail. Generalize a pattern.
4. **AI helps, then we verify**
   Generated code is a draft until we inspect and test it.
5. **Studio**
   Plan a tiny interaction; code only if setup is green.

First-day setup check

Setup status is not a grade. It tells us how to help.

Green

- `npm test` passes.
- `npm run dev` opens the starter game.
- You can start the optional code step.

Yellow

- One command failed.
- You copied the exact error.
- You can still do the paper plan.

Red

- You cannot open the folder or terminal.
- Stop changing things.
- Ask for setup clinic help.

Everyone can complete today's concept work even if setup is yellow or red.

Course promise

We are learning how to build useful, responsible software.

Think clearly

- Name the idea.
- Choose useful details.
- Explain the model.

Build visibly

- Use TypeScript.
- Make small programs work.
- Debug with evidence.

Use AI responsibly

- Ask better questions.
- Check generated code.
- Protect private data.

Class rhythm: explain briefly -> trace together -> build -> share.

Abstraction is the first big move

An abstraction keeps what matters for the job and hides what does not.

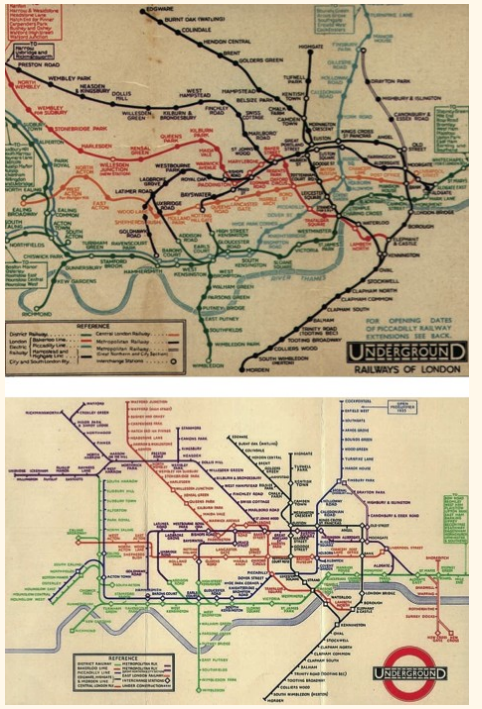
Detail removal

- Keep the useful details.
- Hide what distracts from the task.

Generalization

- Find the repeated pattern.
- Turn it into one rule.

Detail removal in visual form



A map abstracts a city for one task

Your question: what did the map hide, and what did it keep?

Detail removal: from messy world to buildable model

Programming starts by deciding what the program will ignore.

Real world

- People give different inputs.
- Users type surprising input.

→

Buildable model

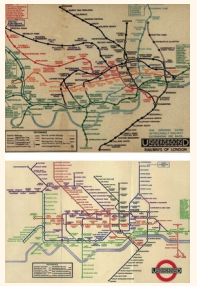
- An input value.
- A validated string.

→

Visible app

- A function returns a result.
- A message explains what happened.

From full detail to the details needed for this task



A transit map optimizes for decisions, not geography

Generalization: one rule beats many copies

The goal is not shorter code. The goal is one clear idea that works for many cases.

```
// Repeated instructions
feedDog();
feedChicken();
feedRabbit();
```

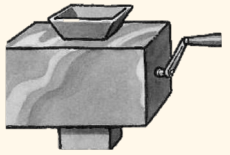
```
// One generalized rule
type Animal = "dog" | "chicken" | "rabbit" |
"sheep";

const feed = (animal: Animal) =>
  `feed the ${animal}`;
```

dog

chicken

rabbit

sheep



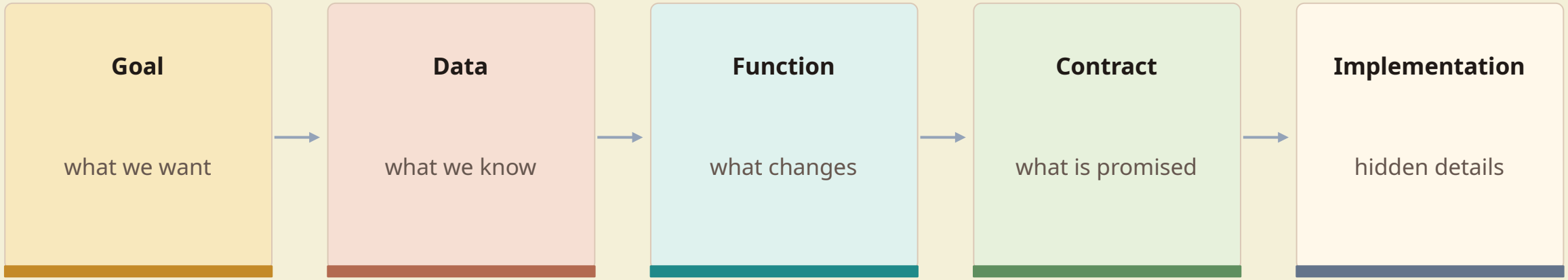
same rule, new input

Why this matters

- The parameter names what changes.
- The type says which inputs make sense.

Programs are abstraction stacks

You do not need every detail at once. You need the layers you can reason about.



Abstraction barrier: use the promise first. Learn the hidden details when they matter.

Interface, specification, contract

An abstraction is strongest when everyone knows what can be relied on.

Interface

- What the outside world can touch.
- Examples: a function name, a parameter, a command.

Specification

- What the interface guarantees.
- Inputs, outputs, timing, errors, and edge cases.

Implementation

- How the promise is kept.
- Details can change if the contract still holds.

interface

compatible input

translation layer

```
const formatName = (name: string): string =>
  name.trim() === "" ? "Anonymous" :
name.trim();
```

First TypeScript trace: transform an input

Read the program as a path from input to output before changing it.

```
type Mood = "focused" | "stuck" | "curious";

const nextStep = (mood: Mood): string => {
  if (mood === "stuck") return "ask a smaller
question";
  if (mood === "curious") return "try one
experiment";
  return "keep building";
};
```

Trace before run

- What inputs are allowed?
- Which branch runs?
- What output comes back?
- What input is impossible?

The studio build

Leave class with a plan you can explain and code only if setup is green.

Required target

- One input.
- One rule.
- One output.
- One edge case.

Pair roles

- Navigator explains the abstraction out loud.
- Driver writes the plan or makes one small code check.
- Switch after each visible result or decision.

Discuss

- What did your function promise?
- What input did your rule allow?
- What is your setup status?

AI is a tool, not an authority

AI can draft an explanation or code snippet. It cannot take responsibility for your program.

AI can help

- Generate ideas and smaller examples.
- Explain a confusing error message.
- Suggest test cases or edge cases.
- Compare two versions of a function.

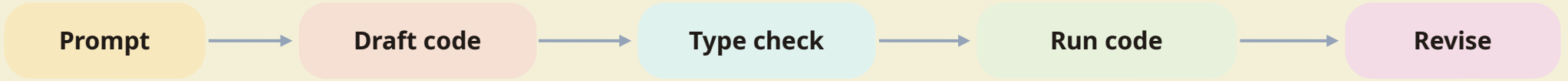
You still own

- Reading every line.
- Checking inputs and outputs.
- Running the code yourself.
- Deciding whether the answer is appropriate.

First rule: generated code is a draft until it survives your types, your tests, and your explanation.

Where AI fits in programming

AI may suggest code for one part while missing the whole idea.



Use AI for a small, specific boundary

- Good: write a function that formats a username.
- Better: include input type, output type, and two edge cases.
- Best: ask for tests before asking for implementation.

Use TypeScript to make the draft checkable

- Types expose vague assumptions.
- Optional values force missing-element handling.
- Small functions are easier to test and explain.

AI collaboration boundaries

The boundary is simple: use help to learn and improve, not to bypass understanding.

Good use

- Explain this concept in a simpler way.
- Give me three practice problems.
- What does this error message mean?
- Suggest edge cases I should test.

Not good use

- Build my whole project for me.
- Write this function so I can submit it.
- Paste private data or API keys.
- Trust generated code without checking.

Activity: AI as pair programmer

The goal is not to get perfect code. The goal is to practice inspection.

```
Prompt:
Ask AI for a TypeScript function that turns
a mood into a next step.
```

In pairs, mark the assumptions

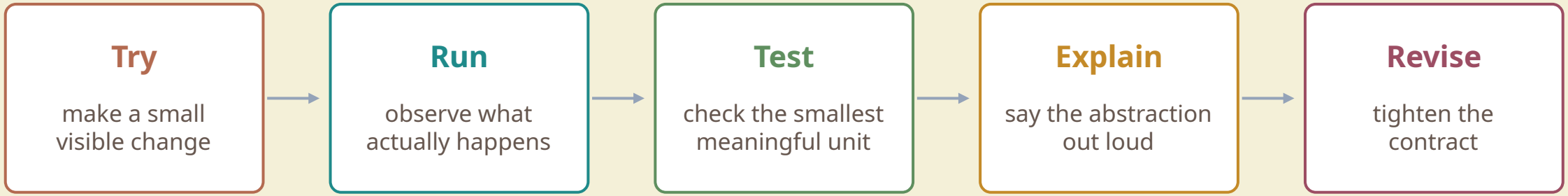
- Which inputs does it allow?
- What output does it promise?
- What edge case is missing?

Improve one thing

- Make the type clearer.
- Move logic into a named function.
- Add one testable edge case.
- Rewrite the prompt more precisely.

Programming is revision

Good code is usually not first-draft code.



What good programmers do

- Treat bugs as information.
- Explain the change.
- Revise one small thing at a time.

Lab plan: practice, build, apply

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

Guided practice

- Locate `START_HERE.md`, the first-run checklist, the pair-work guide, the setup guide for your computer, and `starter/README.md`.
- Run the tool checks for Git, Node.js, and npm.
- Trace input, process, and output for the starter game and one everyday system.

Build target

- Run the starter tests and browser app if setup is green; otherwise capture the exact failed command as a setup puzzle.
- Find the starter rule in `starter/src/game.ts` and trace one input to one output.
- Change one text-only constant or test expectation only if setup is green, then run the matching command.

Project time

- Use `projects/templates/project-choice-card.md` to choose a simple game mechanic for P1.
- List the values the mechanic needs.
- Turn one value into a tiny input-output example the starter code could test later.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Save the passing `npm test` result or the exact setup blocker as a proof object.
- Find where the starter game rule lives in `starter/src/game.ts`.
- Ask AI for a second mechanic, then reduce it to one testable rule.

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 only when the core abstraction is explainable without syntax.

Exit ticket

- One abstraction I can explain.
- My setup status: green, yellow, or red.
- One input, rule, output, and edge case.
- One AI suggestion I would verify before trusting.

A stable interface hides changing implementation

Next class preview

- Expressions, values, and types.
- How programs store and transform values.
- How type errors become design feedback.