L10

AI Game Development & Computer Science Summer Program

Lecture 10: Mutability, Immutability, and References

Mutation changes an existing value

A mutable object can be updated without making a new object.

```
const task = { title: "Trace", done:
false };

task.done = true;
// Same object, changed field.
```

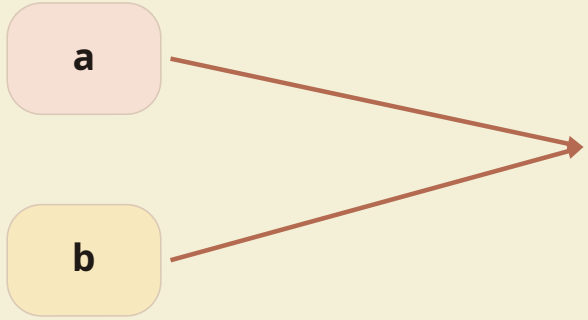
Why it can surprise you

- Any other name pointing to the same object sees the change.
- The change can happen far away from where the value was created.

Mutation is not forbidden. It must be deliberate and easy to trace.

References let two names point to one object

The names are different. The object can be the same.



shared object

- { score: 10 }

Trace question

- If `b.score = 20`, what does `a.score` report?
- Answer: 20, because both names point to the same object.

Copy before changing when you need a new state

A copy makes the change explicit.

```
type Counter = Readonly<{ value: number }>;

const increment = (counter: Counter): Counter => {
  return { ...counter, value: counter.value + 1 };
};
```

What changes

- The old object stays the same.
- The returned object has the updated value.
- Tests can compare before and after.

Nodes are objects too

Mutating a node changes the shared node.

```
const first = { value: "potion" };
const alsoFirst = first;

alsoFirst.next = { value: "key" };

first.next?.value; // "key"
```

Why this matters

- A helper can accidentally change data owned by another part of the program.
- The bug may appear later, after more events or tests.

Safer habit

- Return a new node when the old chain should stay unchanged.

Copying nodes and objects

Use copying patterns that match the data shape.

Goal	Pattern
copy node	{ ...node }
replace value	{ ...node, value: nextValue }
copy object	{ ...task }
change field	{ ...task, done: true }

Important limit

- Spread makes a shallow copy.
- Nested objects can still be shared.
- Copy the level you plan to change.

Readonly types show intent

Readonly tells TypeScript and readers that this function should not mutate input.

```
type Task = Readonly<{ id: string; done: boolean }>;
type ListNode = Readonly<{ value: Task; next?: ListNode
}>;

const isOpen = (node: ListNode): boolean => {
  return !node.value.done;
};
```

What readonly gives you

- A clearer contract.
- Earlier feedback when code tries to mutate input.
- A habit of returning new values.

Test mutation bugs by checking before and after

A good test can prove whether the input changed.

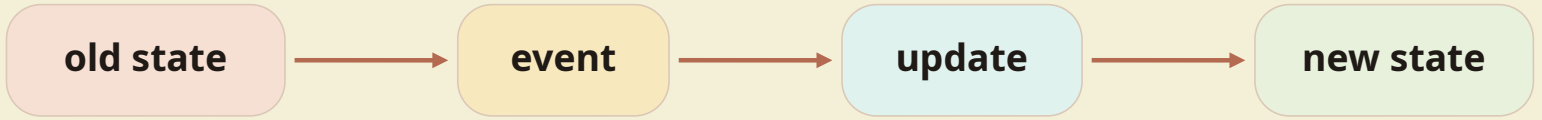
```
const original = { id: "a", done: false };
const updated = markDone(original);

expect(updated.done).toBe(true);
expect(original.done).toBe(false);
```

Test target

- The returned value has the intended update.
- The original value did not change.
- The two values are not the same object.

Immutable updates fit event-driven programs

Each event can produce a new state from the old state.



Reasoning benefit

- You can trace one event at a time.
- You can replay a sequence of events.
- You can test state transitions without hidden changes.

AI can review state updates

Ask AI to look for mutation, then verify the code yourself.

Prompt

- Here is a TypeScript state-update function.
- Identify whether it mutates input or returns a new value.
- Suggest tests that would prove the behavior.

Your review

- Check every assignment and mutating method.
- Trace whether names share the same object.
- Write the before-and-after test yourself.

AI can point at risk. The test proves the behavior.

Studio build: safe task-node update

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

const markCurrentDone = (node: ListNode): ListNode => {
  return {
    ...node,
    value: { ...node.value, done: true },
  };
};
```

Build order

- Write the mutation bug version.
- Write a before-and-after test.
- Fix with node and object copies.
- Add one case for an unchanged next node.

Lab plan: practice, build, apply

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

Guided practice

- Predict object identity and shared references.
- Trace mutation versus copied updates.
- Explain why a state update is safe.

Build target

- Fix a mutation bug by returning new state.
- Write a test that proves old state stayed unchanged.

Project time

- Polish P1 state updates.
- Prepare a short demo and proof objects.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Add a history chain.
- Freeze a sample object and find mutation failures.
- Ask AI to spot shared-state risks.

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 predict which names share which objects.

Exit ticket

- One mutation and its visible effect.
- One example of two names sharing one object.
- One copy pattern for nodes or objects.
- One test that proves input was not mutated.
- One AI-suggested risk you verified.

Next class preview

- Recursion.
- Self-similar problems.
- Functions that solve smaller versions of the same task.