# UNO Assignment 1 — plan + code skeleton (fixed copy)

Below are **known-good, copy/paste-safe** file contents to fix the TypeScript errors you saw. Replace the files verbatim. Then run npm test again and share the first failing test (if any).

## src/model/deck.ts

// src/model/deck.ts  
  
export type Color = 'BLUE' | 'GREEN' | 'RED' | 'YELLOW'  
export const colors: Readonly<Color[]> = ['BLUE', 'GREEN', 'RED', 'YELLOW'] as const  
  
export type Numbered = 0|1|2|3|4|5|6|7|8|9  
export type CardType = 'NUMBERED' | 'SKIP' | 'REVERSE' | 'DRAW' | 'WILD' | 'WILD DRAW'  
  
export type NumberedCard = { type: 'NUMBERED', color: Color, number: Numbered }  
export type ActionCard = { type: 'SKIP'|'REVERSE'|'DRAW', color: Color }  
export type WildCard = { type: 'WILD' }  
export type WildDraw4Card = { type: 'WILD DRAW' }  
export type Card = NumberedCard | ActionCard | WildCard | WildDraw4Card  
  
export type CardMemento = { type: CardType, color?: Color, number?: Numbered }  
  
export type Shuffler<T> = (xs: T[]) => void  
  
export interface Deck {  
 readonly size: number  
 shuffle(s: Shuffler<Card>): void  
 deal(): Card | undefined  
 peek(): Card | undefined  
 top(): Card | undefined  
 toMemento(): CardMemento[]  
}  
  
const clone = <T>(xs: T[]) => xs.slice()  
  
class ArrayDeck implements Deck {  
 private cards: Card[]  
 constructor(cards: Card[]) { this.cards = cards }  
 get size() { return this.cards.length }  
 shuffle(s: Shuffler<Card>) { s(this.cards) }  
 deal() { return this.cards.shift() }  
 peek() { return this.cards[0] }  
 top() { return this.cards[0] }  
 toMemento(): CardMemento[] { return this.cards.map(c => ({...c})) }  
}  
  
export const hasColor = (c: Card, color: Color) => 'color' in c && c.color === color  
export const hasNumber = (c: Card, n: Numbered) => c.type === 'NUMBERED' && c.number === n  
  
const makeStandardCards = (): Card[] => {  
 const cards: Card[] = []  
 for (const color of colors) {  
 cards.push({ type: 'NUMBERED', color, number: 0 })  
 for (let n = 1 as Numbered; n <= 9; n = (n + 1) as Numbered) {  
 cards.push({ type: 'NUMBERED', color, number: n })  
 cards.push({ type: 'NUMBERED', color, number: n })  
 }  
 cards.push({ type: 'SKIP', color }); cards.push({ type: 'SKIP', color })  
 cards.push({ type: 'REVERSE', color }); cards.push({ type: 'REVERSE', color })  
 cards.push({ type: 'DRAW', color }); cards.push({ type: 'DRAW', color })  
 }  
 for (let i = 0; i < 4; i++) cards.push({ type: 'WILD' })  
 for (let i = 0; i < 4; i++) cards.push({ type: 'WILD DRAW' })  
 return cards  
}  
  
export const createStandardDeck = (): Deck => {  
 return new ArrayDeck(makeStandardCards())  
}  
  
export const deckFromMemento = (cards: CardMemento[]): Deck => {  
 const parsed: Card[] = cards.map(raw => {  
 if (raw.type === 'WILD') return { type: 'WILD' }  
 if (raw.type === 'WILD DRAW') return { type: 'WILD DRAW' }  
 if (raw.type === 'NUMBERED') {  
 if (raw.color === undefined || raw.number === undefined) throw new Error('Invalid NUMBERED memento')  
 return { type: 'NUMBERED', color: raw.color, number: raw.number }  
 }  
 if (raw.type === 'SKIP' || raw.type === 'REVERSE' || raw.type === 'DRAW') {  
 if (raw.color === undefined) throw new Error('Missing color for action card')  
 return { type: raw.type, color: raw.color }  
 }  
 throw new Error('Unknown card type')  
 })  
 return new ArrayDeck(parsed)  
}  
  
// Expose a narrow way to recognize an ArrayDeck instance if callers need advanced ops later  
export const asArrayDeck = (d: Deck): any => d as any  
``  
  
---  
  
## src/model/round.ts  
```ts  
// src/model/round.ts  
import { colors, createStandardDeck, deckFromMemento, type Card, type Color, type Deck, type Shuffler } from './deck'  
  
export type Direction = 'clockwise' | 'counterclockwise'  
  
export type RoundConfig = {  
 players: string[]  
 dealer: number  
 cardsPerPlayer?: number  
 shuffler?: Shuffler<Card>  
}  
  
export type RoundMemento = {  
 players: string[]  
 hands: Card[][]  
 drawPile: ReturnType<Deck['toMemento']>  
 discardPile: ReturnType<Deck['toMemento']>  
 currentColor: Color  
 currentDirection: Direction  
 dealer: number  
 playerInTurn: number | undefined  
}  
  
export type EndEvent = { winner: number }  
export type EndListener = (e: EndEvent) => void  
  
export interface Round {  
 readonly playerCount: number  
 player(i: number): string  
 playerHand(i: number): Readonly<Card[]>  
 drawPile(): Deck  
 discardPile(): Deck  
 playerInTurn(): number | undefined  
 dealer: number  
 canPlay(i: number): boolean  
 canPlayAny(): boolean  
 play(i: number, chosenColor?: Color): void  
 draw(): void  
 sayUno(player: number): void  
 catchUnoFailure(args: { accuser: number, accused: number }): boolean  
 hasEnded(): boolean  
 winner(): number | undefined  
 onEnd(l: EndListener): void  
 toMemento(): RoundMemento  
}  
  
const mod = (a: number, n: number) => ((a % n) + n) % n  
  
const pointsFor = (c: Card): number =>  
 c.type === 'NUMBERED' ? c.number : (c.type === 'WILD' || c.type === 'WILD DRAW' ? 50 : 20)  
  
class RoundImpl implements Round {  
 players: string[]  
 dealer: number  
 private hands: Card[][] = []  
 private drawCards: Card[] = []  
 private discardCards: Card[] = [] // top is index 0  
 private curColor: Color  
 private dir: Direction = 'clockwise'  
 private turn: number | undefined  
 private ended = false  
 private theWinner: number | undefined  
 private listeners: EndListener[] = []  
  
 // UNO window  
 private unoOpen = false  
 private unoTarget: number | undefined  
 private unoSaid = false  
  
 constructor(cfg: RoundConfig)  
 constructor(m: RoundMemento, \_shuffler?: Shuffler<Card>)  
 constructor(arg: RoundConfig | RoundMemento, \_shuffler?: Shuffler<Card>) {  
 if ('hands' in arg) {  
 const m = arg as RoundMemento  
 this.players = m.players.slice()  
 this.dealer = m.dealer  
 this.hands = m.hands.map(h => h.slice())  
 this.drawCards = m.drawPile.map(c => c as Card)  
 this.discardCards = m.discardPile.map(c => c as Card)  
 this.curColor = m.currentColor  
 this.dir = m.currentDirection  
 this.turn = m.playerInTurn  
 return  
 }  
  
 const cfg = arg as RoundConfig  
 if (cfg.players.length < 2) throw new Error('Need at least 2 players')  
 const n = cfg.players.length  
 const k = cfg.cardsPerPlayer ?? 7  
 this.players = cfg.players.slice()  
 this.dealer = cfg.dealer  
  
 // Build deck and shuffle ONCE before dealing  
 const deck = createStandardDeck()  
 if (cfg.shuffler) deck.shuffle(cfg.shuffler)  
  
 // Deal 7 cards per player from the top of the deck (sequentially per player)  
 this.hands = Array.from({ length: n }, () => [])  
 for (let p = 0; p < n; p++) {  
 for (let j = 0; j < k; j++) {  
 const c = deck.deal()!  
 this.hands[p].push(c)  
 }  
 }  
  
 // Flip first non-wild card to discard; reshuffle remaining if wild  
 let first = deck.deal()!  
 while (first.type === 'WILD' || first.type === 'WILD DRAW') {  
 if (!cfg.shuffler) break  
 deck.shuffle(cfg.shuffler)  
 first = deck.deal()!  
 }  
  
 this.discardCards = [first]  
  
 // Remaining cards become the draw pile (in current order)  
 this.drawCards = []  
 while (true) {  
 const c = deck.deal()  
 if (!c) break  
 this.drawCards.push(c)  
 }  
  
 this.curColor = 'color' in first ? first.color : colors[0]  
  
 // Starting player and initial effect  
 const left = mod(this.dealer + 1, n)  
 if (first.type === 'REVERSE') {  
 this.dir = 'counterclockwise'  
 this.turn = mod(this.dealer - 1, n)  
 } else if (first.type === 'SKIP') {  
 this.turn = mod(this.dealer + 2, n)  
 } else if (first.type === 'DRAW') {  
 this.giveCards(left, 2)  
 this.turn = mod(this.dealer + 2, n)  
 } else {  
 this.turn = left  
 }  
 }  
  
 get playerCount() { return this.players.length }  
 player(i: number) { return this.players[i] }  
 playerHand(i: number) { return this.hands[i] }  
  
 drawPile(): Deck { return deckFromMemento(this.drawCards.map(c => ({...c}))) }  
 discardPile(): Deck { return deckFromMemento(this.discardCards.map(c => ({...c}))) }  
  
 playerInTurn() { return this.turn }  
  
 private playableAgainst(card: Card, top: Card, currentColor: Color, allowTypeMatch: boolean): boolean {  
 if (top.type === 'WILD' || top.type === 'WILD DRAW') {  
 if ('color' in card) return card.color === currentColor  
 return true  
 }  
 if (card.type === 'NUMBERED') {  
 if (top.type === 'NUMBERED') return card.color === currentColor || card.number === top.number  
 return card.color === currentColor  
 }  
 if (card.type === 'SKIP' || card.type === 'REVERSE' || card.type === 'DRAW') {  
 if (allowTypeMatch && top.type === card.type) return true  
 return card.color === currentColor  
 }  
 return true  
 }  
  
 canPlay(i: number): boolean {  
 if (this.ended || this.turn === undefined) return false  
 const hand = this.hands[this.turn]  
 if (i < 0 || i >= hand.length) return false  
 const card = hand[i]  
 const top = this.discardCards[0]  
  
 if (card.type === 'WILD') return true  
 if (card.type === 'WILD DRAW') {  
 return !hand.some((c, idx) => idx !== i && this.playableAgainst(c, top, this.curColor, true))  
 }  
 return this.playableAgainst(card, top, this.curColor, true)  
 }  
  
 canPlayAny(): boolean {  
 if (this.ended || this.turn === undefined) return false  
 return this.hands[this.turn].some((\_, i) => this.canPlay(i))  
 }  
  
 play(i: number, chosenColor?: Color): void {  
 if (!this.canPlay(i)) throw new Error('Illegal play')  
 const p = this.turn!  
 const card = this.hands[p].splice(i, 1)[0]  
 this.discardCards.unshift(card)  
  
 if (card.type === 'WILD' || card.type === 'WILD DRAW') {  
 if (!chosenColor) throw new Error('Chosen color required for wild')  
 this.curColor = chosenColor  
 } else if ('color' in card) {  
 this.curColor = card.color  
 }  
  
 this.unoOpen = this.hands[p].length === 1  
 this.unoTarget = this.unoOpen ? p : undefined  
 this.unoSaid = false  
  
 this.advanceAfterPlay(card)  
  
 if (this.hands[p].length === 0) {  
 this.ended = true  
 this.theWinner = p  
 this.turn = undefined  
 this.unoOpen = false  
 this.listeners.forEach(l => l({ winner: p }))  
 }  
 }  
  
 private advanceAfterPlay(card: Card) {  
 const n = this.playerCount  
 const step = this.dir === 'clockwise' ? 1 : -1  
 const next = (x: number, s = step) => mod(x + s, n)  
  
 if (card.type === 'SKIP') { this.turn = next(this.turn!); this.turn = next(this.turn!); return }  
 if (card.type === 'REVERSE') {  
 if (n === 2) { this.turn = next(this.turn!); return }  
 this.dir = this.dir === 'clockwise' ? 'counterclockwise' : 'clockwise'  
 this.turn = next(this.turn!)  
 return  
 }  
 if (card.type === 'DRAW') { const victim = next(this.turn!); this.giveCards(victim, 2); this.turn = next(victim); return }  
 if (card.type === 'WILD DRAW') { const victim = next(this.turn!); this.giveCards(victim, 4); this.turn = next(victim); return }  
 this.turn = next(this.turn!)  
 }  
  
 draw(): void {  
 if (this.ended || this.turn === undefined) throw new Error('Round ended')  
 this.unoOpen = false  
  
 if (this.drawCards.length === 0) this.rebuildDrawFromDiscard()  
 const card = this.drawCards.shift()  
 if (!card) return  
  
 const p = this.turn  
 this.hands[p].push(card)  
  
 const top = this.discardCards[0]  
 const playable = this.playableAgainst(card, top, this.curColor, true) || card.type === 'WILD' || card.type === 'WILD DRAW'  
 if (!playable) {  
 const n = this.playerCount  
 const step = this.dir === 'clockwise' ? 1 : -1  
 this.turn = mod(this.turn + step, n)  
 }  
 }  
  
 private giveCards(player: number, k: number) {  
 for (let i = 0; i < k; i++) {  
 if (this.drawCards.length === 0) this.rebuildDrawFromDiscard()  
 const c = this.drawCards.shift()  
 if (c) this.hands[player].push(c)  
 }  
 }  
  
 private rebuildDrawFromDiscard() {  
 const top = this.discardCards.shift()  
 const rest = this.discardCards.splice(0)  
 if (rest.length === 0) { if (top) this.discardCards.unshift(top); return }  
 // naive shuffle  
 rest.sort(() => Math.random() - 0.5)  
 this.drawCards.push(...rest)  
 if (top) this.discardCards.unshift(top)  
 }  
  
 sayUno(player: number) { if (this.unoOpen && this.unoTarget === player) this.unoSaid = true }  
  
 catchUnoFailure({ accuser, accused }: { accuser: number, accused: number }): boolean {  
 if (!this.unoOpen || this.unoTarget !== accused) return false  
 if (this.unoSaid) return false  
 this.giveCards(accused, 4)  
 this.unoOpen = false  
 return true  
 }  
  
 hasEnded() { return this.ended }  
 winner() { return this.theWinner }  
 onEnd(l: EndListener) { this.listeners.push(l) }  
  
 toMemento(): RoundMemento {  
 return {  
 players: this.players.slice(),  
 hands: this.hands.map(h => h.slice()),  
 drawPile: this.drawCards.map(c => ({...c})),  
 discardPile: this.discardCards.map(c => ({...c})),  
 currentColor: this.curColor,  
 currentDirection: this.dir,  
 dealer: this.dealer,  
 playerInTurn: this.turn  
 }  
 }  
}  
  
export type { RoundImpl }  
export const createRound = (cfg: RoundConfig): Round => new RoundImpl(cfg)  
export const createRoundFromMemento = (m: RoundMemento, \_shuffler?: Shuffler<Card>): Round => new RoundImpl(m, \_shuffler)  
export const pointsForCard = pointsFor  
``  
  
---  
  
## src/model/uno.ts  
```ts  
// src/model/uno.ts  
import type { Card } from './deck'  
import { createRound, createRoundFromMemento, type Round, type RoundMemento } from './round'  
  
export type GameMemento = {  
 players: string[]  
 targetScore: number  
 scores: number[]  
 currentRound?: RoundMemento  
}  
  
export interface Game {  
 readonly playerCount: number  
 player(i: number): string  
 score(i: number): number  
 winner(): number | undefined  
 currentRound(): Round | undefined  
 toMemento(): GameMemento  
}  
  
export type GameConfig = {  
 players?: string[]  
 targetScore?: number  
}  
  
class GameImpl implements Game {  
 players: string[]  
 target: number  
 private scores: number[]  
 private roundInst: Round | undefined  
 private theWinner: number | undefined  
  
 constructor(cfg: GameConfig)  
 constructor(m: GameMemento)  
 constructor(arg: GameConfig | GameMemento) {  
 if ('scores' in arg) {  
 const m = arg as GameMemento  
 this.players = m.players.slice()  
 this.target = m.targetScore  
 this.scores = m.scores.slice()  
 this.roundInst = m.currentRound ? createRoundFromMemento(m.currentRound) : undefined  
 const w = this.scores.findIndex(s => s >= this.target)  
 this.theWinner = w >= 0 ? w : undefined  
 } else {  
 const cfg = arg as GameConfig  
 this.players = cfg.players ?? ['A','B']  
 this.target = cfg.targetScore ?? 500  
 this.scores = new Array(this.players.length).fill(0)  
 }  
 }  
  
 get playerCount() { return this.players.length }  
 player(i: number) { return this.players[i] }  
 score(i: number) { return this.scores[i] }  
 winner() { return this.theWinner }  
 currentRound() { return this.roundInst }  
  
 toMemento(): GameMemento {  
 return {  
 players: this.players.slice(),  
 targetScore: this.target,  
 scores: this.scores.slice(),  
 currentRound: this.roundInst?.toMemento()  
 }  
 }  
}  
  
export const createGame = (cfg: GameConfig): Game => new GameImpl(cfg)  
export const createGameFromMemento = (m: GameMemento): Game => new GameImpl(m)  
``  
  
---  
  
## \_\_test\_\_/utils/test\_adapter.ts (adapter)  
```ts  
// \_\_test\_\_/utils/test\_adapter.ts  
import { standardShuffler, type Randomizer, type Shuffler, memoizingShuffler } from '../../src/utils/random\_utils'  
import { type Card, deckFromMemento, createStandardDeck } from '../../src/model/deck'  
import { createRound as \_createRound, createRoundFromMemento as \_createRoundFromMemento, type Round, type RoundMemento } from '../../src/model/round'  
import { createGame as \_createGame, createGameFromMemento as \_createGameFromMemento, type Game, type GameMemento } from '../../src/model/uno'  
  
export function createInitialDeck() { return createStandardDeck() }  
export function createDeckFromMemento(cards: Record<string, string | number>[]) { return deckFromMemento(cards as any) }  
  
export type HandConfig = { players: string[], dealer: number, cardsPerPlayer?: number, shuffler?: Shuffler<Card> }  
export function createRound(props: HandConfig): Round { return \_createRound(props) }  
export function createRoundFromMemento(m: RoundMemento, shuffler?: Shuffler<Card>): Round { return \_createRoundFromMemento(m, shuffler) }  
  
export type GameConfig = { players: string[], targetScore: number, randomizer?: Randomizer, shuffler?: Shuffler<Card>, cardsPerPlayer?: number }  
export function createGame(props: Partial<GameConfig>): Game {  
 return \_createGame({ players: props.players, targetScore: props.targetScore })  
}  
export function createGameFromMemento(m: GameMemento, \_randomizer?: Randomizer, \_shuffler?: Shuffler<Card>): Game {  
 return \_createGameFromMemento(m)  
}  
  
export { standardShuffler, memoizingShuffler }

### After pasting

1. Save all files.
2. Run: npm run clean && npm test (or just npm test).
3. Tell me the **first failing test name and message** and I’ll close the gap.