TypeScript Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a type-safe compose function in TypeScript that can handle multiple functions

Here's an implementation of a type-safe compose function:

```
type Func = (x: T) => T;
const compose = (...fns: Func[]): Func =>
  (x: T) => fns.reduceRight((y, f) => f(y), x);
```

Key points:

- Uses generics for type safety
- Maintains type consistency across function chain
- Handles arbitrary number of functions

2. Create a type-safe deep readonly utility type

```
type DeepReadonly = {
  readonly [P in keyof T]: T[P] extends object
    ? DeepReadonly
    : T[P]
};
```

Usage example:

```
interface Config {
  data: { id: number; }
}
type ReadonlyConfig = DeepReadonly;
```

3. Implement a type-safe event emitter class

```
class TypedEmitter> {
  private listeners = new Map();
  on(event: K, fn: (data: T[K]) => void) {
    const fns = this.listeners.get(event) || [];
    this.listeners.set(event, [...fns, fn]);
  }
```

Benefits:

- Type-safe event names and payloads
- Autocomplete support
- Compile-time type checking

4. Create a type-safe builder pattern implementation

```
class Builder {
  private obj: Partial = {};
  set(key: K, value: T[K]): this {
    this.obj[key] = value;
    return this;
  }
  build(): T { return this.obj as T; }
}
```

Key features:

- Fluent interface
- Type-safe property setting
- Compile-time validation

5. Implement a type-safe memoization decorator

```
function Memoize any>() {
  return function(target: any, key: string, descriptor: PropertyDescriptor) {
    const cache = new Map>();
    const original = descriptor.value;
    descriptor.value = function(...args: Parameters) {
      const key = JSON.stringify(args);
      if (!cache.has(key)) cache.set(key, original.apply(this, args));
      return cache.get(key);
    };
  };
}
```

6. Create a type-safe dependency injection container

```
class Container {
  private services = new Map();
  register(token: string, provider: () => T): void {
    this.services.set(token, provider);
  }
  resolve(token: string): T {
    return this.services.get(token)();
  }
}
```

Features:

- Type-safe service registration
- Lazy initialization
- Dependency resolution

7. Implement a type-safe state machine

```
type State = {
  [K in S]: { on: { [K in E]?: S } }
};
class StateMachine {
  constructor(private state: S, private config: State) {}
  transition(event: E): S {
    this.state = this.config[this.state].on[event] || this.state;
    return this.state;
  }
}
```

8. Create a type-safe proxy wrapper

```
function createProxy(target: T): T {
  return new Proxy(target, {
    get: (obj, prop: keyof T) => {
      console.log(`Accessing ${String(prop)}`);
      return obj[prop];
    }
  });
}
```

Features:

- Type-safe property access
- Runtime property tracking
- Transparent wrapping

9. Implement a type-safe observable pattern

```
class Observable {
  private observers: ((value: T) => void)[] = [];
  subscribe(observer: (value: T) => void): () => void {
    this.observers.push(observer);
    return () => this.observers =
      this.observers.filter(obs => obs !== observer);
  }
}
```

Key aspects:

- Type-safe value propagation
- Subscription management
- Cleanup handling

10. Create a type-safe validation decorator

```
function Validate(validator: (value: T) => boolean) {
  return function(target: any, propertyKey: string) {
    let value: T;
    Object.defineProperty(target, propertyKey, {
      set(v: T) { if (validator(v)) value = v; },
      get() { return value; }
    });
  };
}
```

Benefits:

- Compile-time type checking
- Runtime validation
- Property decoration

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement a generic Stack data structure in TypeScript with push, pop, and peek operations

Solution:

```
class Stack {
  private items: T[] = [];
  push(item: T): void { this.items.push(item); }
  pop(): T | undefined { return this.items.pop(); }
  peek(): T | undefined { return this.items[this.items.length - 1]; }
  isEmpty(): boolean { return this.items.length === 0; }
}
```

Time Complexity:

- Push: O(1)
- Pop: O(1)
- Peek: O(1)

2. Create a function that finds pairs of numbers in an array that sum to a target value using TypeScript

Solution:

```
function findPairs(arr: number[], target: number): [number, number][] {
  const pairs: [number, number][] = [];
  const seen = new Set();
  arr.forEach(num => {
    if (seen.has(target - num)) pairs.push([num, target - num]);
    seen.add(num);
  });
  return pairs;
}
```

Time Complexity: O(n) where n is array length

3. Implement an LRU (Least Recently Used) Cache with a fixed capacity in TypeScript

Solution:

```
class LRUCache {
  private cache = new Map();
  constructor(private capacity: number) {}
  get(key: K): V | undefined {
    const value = this.cache.get(key);
    if (value) { this.cache.delete(key); this.cache.set(key, value); }
    return value;
  }
  put(key: K, value: V): void {
    if (this.cache.size >= this.capacity) this.cache.delete(this.cache.keys().next().value);
    this.cache.set(key, value);
  }
}
```

Time Complexity:

- Get: O(1)

- Put: O(1)

4. Implement a function to find the maximum sum of any contiguous subarray using TypeScript (Kadane's Algorithm)

Solution:

```
function maxSubArraySum(arr: number[]): number {
  let maxSoFar = arr[0], maxEndingHere = arr[0];
  for (let i = 1; i < arr.length; i++) {
    maxEndingHere = Math.max(arr[i], maxEndingHere + arr[i]);
    maxSoFar = Math.max(maxSoFar, maxEndingHere);
  }
  return maxSoFar;
}
```

Time Complexity: O(n) where n is array length

5. Create a generic Queue implementation using two Stacks in TypeScript

Solution:

```
class Queue {
  private stack1: T[] = [];
  private stack2: T[] = [];
  enqueue(item: T): void { this.stack1.push(item); }
  dequeue(): T | undefined {
    if (!this.stack2.length) while(this.stack1.length) this.stack2.push(this.stack1.pop()!);
    return this.stack2.pop();
  }
}
```

Time Complexity:

- Enqueue: O(1)
- Dequeue: Amortized O(1)

6. Implement a function to detect if a linked list has a cycle using TypeScript

Solution:

```
interface ListNode { val: number; next: ListNode | null; }
function hasCycle(head: ListNode | null): boolean {
  let slow = head, fast = head;
  while (fast && fast.next) {
    slow = slow!.next;
    fast = fast.next.next;
    if (slow === fast) return true;
  }
  return false;
}
```

Time Complexity: O(n) where n is the number of nodes

7. Create a function to implement binary search on a sorted array in TypeScript

Solution:

```
function binarySearch(arr: T[], target: T): number {
  let left = 0, right = arr.length - 1;
  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    arr[mid] < target ? left = mid + 1 : right = mid - 1;
  }
  return -1;
}
```

Time Complexity: O(log n) where n is array length

8. Implement a function to find the first non-repeating character in a string using TypeScript

Solution:

```
function firstNonRepeating(str: string): string {
  const charMap = new Map();
  for (const char of str) charMap.set(char, (charMap.get(char) || 0) + 1);
  for (const char of str) if (charMap.get(char) === 1) return char;
  return '';
}
```

Time Complexity: O(n) where n is string length

9. Create a MinHeap data structure implementation in TypeScript

Solution:

```
class MinHeap {
  private heap: number[] = [];
  insert(val: number): void {
    this.heap.push(val);
    this.bubbleUp(this.heap.length - 1);
  }
  extractMin(): number | undefined {
    if (this.heap.length === 0) return undefined;
    const min = this.heap[0];
    this.heap[0] = this.heap.pop()!;
    this.bubbleDown(0);
    return min;
  }
}
```

Time Complexity:

- Insert: O(log n)
- ExtractMin: O(log n)

10. Implement a function to reverse a linked list in-place using TypeScript

Solution:

```
interface ListNode { val: number; next: ListNode | null; }
function reverseList(head: ListNode | null): ListNode | null {
  let prev: ListNode | null = null;
  while (head) {
    const next = head.next;
    head.next = prev;
    prev = head;
    head = next;
  }
  return prev;
}
```

Time Complexity: O(n) where n is the number of nodes

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly

Key Requirements:

- Generate unique short URLs
- Redirect to original URL
- High availability
- Low latency

System Components:

- **Load Balancer:** Distribute traffic across application servers
- **Application Servers:** Handle URL shortening and redirection logic
- **Database:** Store URL mappings (NoSQL like Cassandra for scale)
- **Cache Layer:** Redis for frequently accessed URLs

URL Generation:

```
function generateShortUrl(longUrl: string): string {
  const hash = crypto.createHash('md5').update(longUrl).digest('hex');
  return hash.substring(0, 8);
}
```

Scale Considerations:

- Partition data by URL hash
- Use CDN for global access
- Implement rate limiting

2. Design a real-time chat system supporting millions of users

Core Requirements:

- Real-time message delivery
- Support for 1:1 and group chats
- Message persistence
- Online/offline status

Architecture Components:

- **WebSocket Server:** Handle real-time connections
- **Message Queue:** Kafka/RabbitMQ for message routing
- **Database:** MongoDB for messages, Redis for presence

Message Flow:

```
interface Message {
  id: string;
  from: string;
  to: string;
  content: string;
  timestamp: number;
}
```

Scaling Strategy:

- Shard by chat room/user
- Use pub/sub for message broadcasting

3. Design a social media feed system like Twitter

Key Features:

- Post creation and retrieval
- Follow/unfollow functionality
- News feed generation
- Trending topics

System Design:

- **Write Path:** Fan-out on write for followers
- **Read Path:** Pull model for inactive users
- **Storage:** Distributed database (Cassandra)

Feed Generation:

```
interface Post {
  id: string;
  userId: string;
  content: string;
  timestamp: number;
  likes: number;
}
```

Optimization Techniques:

- Cache hot users
- Implement infinite scroll
- Use materialized views for feeds

4. Design a distributed task scheduler system

Requirements:

- Schedule one-time and recurring tasks
- Fault tolerance
- Task prioritization
- Monitoring and logging

Components:

- **Master Node:** Task distribution and scheduling
- **Worker Nodes:** Task execution
- **Queue:** Redis/RabbitMQ for task storage

Task Definition:

```
interface Task {
  id: string;
  type: 'oneTime' | 'recurring';
  schedule: string; // cron expression
  payload: any;
}
```

Scaling Considerations:

- Leader election for master
- Partition tasks by time

5. Design a distributed caching system like Redis

Core Features:

- Key-value storage
- Data partitioning
- Replication
- Eviction policies

Architecture:

- **Cache Nodes:** Store data in memory
- **Proxy Layer:** Handle client requests
- **Consistency Protocol:** Eventual consistency

Cache Entry:

```
interface CacheEntry {
  key: string;
  value: any;
  ttl: number;
  lastAccessed: number;
}
```

Design Considerations:

- Consistent hashing
- Write-through/write-behind
- Cache invalidation

6. Design a rate limiting system for an API gateway

Requirements:

- Limit requests per user/IP
- Multiple time windows
- Distributed system support

Algorithms:

- **Token Bucket**
- **Sliding Window**
- **Fixed Window Counter**

Implementation:

```
class RateLimiter {
  check(key: string, limit: number, window: number): boolean {
    const count = this.getCount(key, window);
    return count < limit;
  }
}
```

Storage Options:

- Redis for counters
- Local cache for performance

7. Design a distributed job queue system

Key Features:

- Job scheduling and execution
- Priority queues
- Retry mechanism
- Dead letter queue

Components:

- **Queue Manager:** Handle job distribution
- **Workers:** Process jobs

- **Storage:** Persist job state

Job Structure:

```
interface Job {
  id: string;
  priority: number;
  payload: any;
  attempts: number;
  status: JobStatus;
}
```

Scaling Strategy:

- Horizontal scaling of workers
- Queue partitioning
- Back-pressure handling

8. Design a notification service supporting multiple channels

Requirements:

- Support email, SMS, push notifications
- Template management
- Delivery tracking
- Rate limiting

Architecture:

- **API Gateway:** Handle requests
- **Template Service:** Manage templates
- **Provider Services:** Channel-specific logic

Notification:

```
interface Notification {
  userId: string;
  template: string;
  channel: Channel[];
  data: Record;
}
```

Considerations:

- Queue-based processing
- Retry strategies
- Provider fallbacks

9. Design a distributed logging and monitoring system

Core Features:

- Log aggregation
- Real-time processing
- Search capabilities
- Alerting

Components:

- **Collectors:** Gather logs
- **Stream Processor:** Real-time analysis
- **Storage:** Elasticsearch for search

Log Structure:

```
interface LogEntry {
  timestamp: number;
```

~~level: LogLevel;~~
~~service: string;~~
~~message: string;~~
~~metadata: any;~~
~~}~~

~~Scale Considerations:~~

- ~~Log rotation~~
- ~~Index management~~
- ~~Data retention policies~~

~~10. Design a distributed configuration management system~~

~~Requirements:~~

- ~~Configuration versioning~~
- ~~Real-time updates~~
- ~~Access control~~
- ~~Audit logging~~

~~Architecture:~~

- ~~**Config Store:** ZooKeeper/etcd~~
- ~~**Cache Layer:** Local cache~~
- ~~**Change Notification:** Pub/sub~~

~~Config Structure:~~

~~interface Config {~~
~~key: string;~~
~~value: any;~~
~~version: number;~~
~~environment: string;~~
~~}~~

~~Design Considerations:~~

- ~~Consistency model~~
- ~~Cache invalidation~~
- ~~Rollback support~~

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Create a type-safe curry function in TypeScript

Solution:

```
type Curry

 =
  P extends [infer First, ...infer Rest]
    ? (arg: First) => Curry
    : R;

function curry

(
  fn: (...args: P) => R
): Curry {
  return (arg: any) =>
    fn.length <= 1 ? fn(arg) : curry((fn as any).bind(null, arg));
}
```

2. Write a TypeScript function to flatten a nested array of any depth

Solution:

```
function flattenArray(arr: (T | T[])[]): T[] {
  return arr.reduce((flat, item) =>
    flat.concat(Array.isArray(item) ? flattenArray(item) : item),
    [] as T[]);
}
```

Key Points:

- Uses generic type T for type safety
- Recursively flattens nested arrays
- Handles arrays of any depth

3. Implement a type-safe debounce function in TypeScript

Solution:

```
function debounce void>(
  fn: T, delay: number): (...args: Parameters) => void {
  let timeoutId: NodeJS.Timeout;
  return (...args: Parameters) => {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn(...args), delay);
  };
}
```

Features:

- Preserves function parameter types
- Uses Parameters utility type
- Properly handles timeout clearing

4. Create a TypeScript type that makes all properties of an interface optional at any depth

Solution:

```
type DeepPartial = {
  [P in keyof T]?: T[P] extends object ? DeepPartial : T[P];
};
```

Usage Example:

```
interface User {
  name: string;
  settings: { theme: string; notifications: boolean; }
}
type PartialUser = DeepPartial;
```

5. Write a TypeScript function to check if a string is a valid palindrome, ignoring spaces and punctuation

Solution:

```
function isPalindrome(str: string): boolean {
  const cleaned = str.toLowerCase().replace(/[^a-z0-9]/g, '');
  return cleaned === cleaned.split('').reverse().join('');
}
```

Test Cases:

- isPalindrome('A man, a plan, a canal: Panama') → true
- isPalindrome('race a car') → false

6. Implement a type-safe event emitter in TypeScript

Solution:

```
type EventMap = { [key: string]: any[] };
class TypedEmitter {
  private listeners = new Map();
  on(event: K, fn: (...args: T[K]) => void) {
    this.listeners.set(event,
      [...(this.listeners.get(event) || []), fn]);
  }}
```

Benefits:

- Type-safe event names and payloads
- Compile-time checking of event handlers

7. Create a TypeScript decorator to measure function execution time

Solution:

```
function measure() {
  return function (target: any, key: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = function (...args: any[]) {
      const start = performance.now();
      const result = original.apply(this, args);
      console.log(`${key} took ${performance.now() - start}ms`);
      return result;
    };
  };
}
```

8. Implement a type-safe memoization function in TypeScript

Solution:

```
function memoize any>(fn: T): T {
  const cache = new Map>();
```

```
  return ((...args: Parameters): ReturnType => {
    const key = JSON.stringify(args);
    if (!cache.has(key)) cache.set(key, fn(...args));
    return cache.get(key)!;
  }) as T;
}
```

9. Write a TypeScript function to implement deep object comparison

Solution:

```
function deepEqual(obj1: any, obj2: any): boolean {
  if (obj1 === obj2) return true;
  if (typeof obj1 !== 'object' || !obj1 || !obj2) return false;
  return Object.keys(obj1).every(key =>
    deepEqual(obj1[key], obj2[key]));
}
```

Features:

- Handles nested objects
- Compares values recursively
- Type-safe comparison

10. Implement a TypeScript utility type that converts string literal types to camelCase

Solution:

```
type ToCamelCase = S extends `${infer P1}_${infer P2}${infer P3}`
  ? `${Lowercase}${Uppercase}${ToCamelCase}`
  : Lowercase;
```

Usage:

- type Result = ToCamelCase<'hello_world'> // 'helloWorld'
- type Result2 = ToCamelCase<'user_first_name'> // 'userFirstName'