Below is a set of questions and answers derived directly from the content you provided for "Week 9.2: Introduction to TypeScript." These focus on the key concepts of TypeScript, including language classifications, its purpose, execution, basic types, interfaces, types, and their differences, ensuring a thorough understanding of the material.

Types of Languages

Q1: What are the characteristics of loosely typed languages?
A1: Loosely typed languages have:

1. **Runtime Type Association**: Types are associated with values at runtime, not during compilation.
2. **Dynamic Type Changes**: Variables can change types during execution, offering flexibility.
3. **Runtime Error Discovery**: Type errors are detected at runtime, potentially causing unexpected behavior.
4. **Examples**: JavaScript, Python, Ruby.

Q2: Why does the C++ code example fail, and what does it demonstrate?
A2: The C++ code fails because:

- `int number = 10; number = "text";` attempts to assign a string to an integer variable, which is not allowed in C++, a statically-typed language.
- This demonstrates that statically-typed languages enforce type consistency at compile-time, catching errors early. Example:

```
int main() {
  int number = 10;
  number = "text"; // Compile-time error
  return 0;
}
```

Q3: What are the features of strongly typed languages?
A3: Strongly typed languages feature:

1. **Compile-Time Enforcement**: Types are checked and enforced during compilation.
2. **Type Safety**: Operations are restricted to compatible types, ensured by the compiler.

3. **Early Error Detection**: Type errors are caught at compile-time, improving reliability.
4. **Examples**: Java, C#, TypeScript.

Q4: Why does the JavaScript code example work, and what does it show?
A4: The JavaScript code works because:

- `let number = 10; number = "text";` is valid due to JavaScript's dynamic typing, allowing type changes at runtime.
- This shows the flexibility of loosely typed languages but highlights the risk of runtime errors. Example:

```
function main() {
  let number = 10;
  number = "text"; // Valid in JavaScript
  return number;
}
```

TypeScript

Q5: Why was TypeScript introduced, and what problem does it address?
A5: TypeScript was introduced by Microsoft to address JavaScript's dynamic typing limitations, which can lead to runtime errors. As a superset of JavaScript, it adds static typing to catch errors during development, enhancing code safety and reliability.

Q6: What are the key features of TypeScript?
A6: Key features include:

1. **Static Typing**: Declares types at compile-time for variables, parameters, and returns.
2. **Compatibility with JavaScript**: All valid JavaScript is valid TypeScript.
3. **Tooling Support**: Offers robust tools like the TypeScript compiler (`tsc`) and IDE enhancements.
4. **Enhanced IDE Experience**: Improves autocompletion, navigation, and refactoring.
5. **Interfaces and Type Declarations**: Defines object shapes and contracts.
6. **Compilation**: Transpiles to JavaScript for runtime compatibility.

Q7: How is TypeScript code executed?

A7: TypeScript execution involves:

1. Writing code in `.ts` or `.tsx` files.
2. Using the TypeScript Compiler (`tsc`) to compile it.
3. The compiler checks types and generates JavaScript (`.js` or `.jsx`).
4. The generated JavaScript runs in any JavaScript environment (e.g., browsers, Node.js).
5. In browsers, it may interact with the DOM.

TypeScript Compiler (`tsc`)

Q8: What is the role of the TypeScript Compiler (`tsc`)?
A8: The TypeScript Compiler (`tsc`):

- Transpiles TypeScript code to JavaScript.
- Performs type checking and error reporting.
- Emits JavaScript files based on `tsconfig.json` settings.
- Is installed via npm and run from the command line.

Q9: What are some alternative tools to `tsc` mentioned in the content?
A9: Alternatives include:

1. **esbuild**: A fast JavaScript bundler and minifier with TypeScript support.
2. **swc (Speedy Web Compiler)**: A high-performance JavaScript/TypeScript compiler.

Setting up a TypeScript Node.js Application

Q10: How do you set up a basic TypeScript Node.js application according to the content?
A10: Steps:

1. Install TypeScript globally: `npm install -g typescript` .
2. Create a project: `mkdir node-app; cd node-app; npm init -y; npx tsc --init` .
3. Write a TypeScript file (e.g., `a.ts`): `const x: number = 1; console.log(x);` .
4. Compile with `tsc -b` , generating `index.js` .

5. Test a type error (e.g., `x = "harkirat"`) and recompile to see the error.

Q11: What happens when you introduce a type error in the TypeScript file?
A11: When a type error occurs (e.g., `let x: number = 1; x = "harkirat";`):

- `tsc -b` detects the mismatch and reports an error in the console.
- No `index.js` file is generated, demonstrating TypeScript's compile-time type checking.

Basic Types in TypeScript

Q12: What are the basic types in TypeScript, with examples?
A12: Basic types include:

1. **Number**: `let age: number = 25;`
2. **String**: `let name: string = "John";`
3. **Boolean**: `let isStudent: boolean = true;`
4. **Null**: `let myVar: null = null;`
5. **Undefined**: `let myVar: undefined = undefined;`

Problems and Code Implementation

Q13: How does the `greet` function demonstrate typing function arguments?
A13: The `greet` function:

- Takes a `firstName: string` parameter and returns `void`.
- Logs a greeting. Example:

```
function greet(firstName: string): void {
  console.log("Hello " + firstName);
}
greet("harkirat"); // Outputs: Hello harkirat
```

Q14: How does the `sum` function show return type assignment?
A14: The `sum` function:

- Takes two `number` parameters and returns a `number`.
- Adds and returns their sum. Example:

```
function sum(a: number, b: number): number {
  return a + b;
}
console.log(sum(2, 3)); // Outputs: 5
```

Q15: How does the `isLegal` function illustrate type inference?

A15: The `isLegal` function:

- Takes an `age: number` and returns a `boolean` based on `age > 18`.
- TypeScript infers the return type as `boolean`. Example:

```
function isLegal(age: number): boolean {
  if (age > 18) return true;
  else return false;
}
console.log(isLegal(22)); // Outputs: true
```

Q16: How does the `delayedCall` function use functions as parameters?

A16: The `delayedCall` function:

- Takes a function `fn: () => void` and executes it after 1 second using `setTimeout`.
- Returns `void`. Example:

```
function delayedCall(fn: () => void): void {
  setTimeout(fn, 1000);
}
delayedCall(() => console.log("hi there")); // Logs "hi there" after 1 second
```

The `tsconfig.json` File in TypeScript

Q17: What does the `target` option in `tsconfig.json` control?

A17: The `target` option specifies the ECMAScript version for compiled JavaScript:

- `"es5"` : Generates ES5-compatible code (e.g., `var greet = function...`).
- `"es2020"` : Uses modern syntax (e.g., `const greet = (name) => ...`). Example:

```
{ "compilerOptions": { "target": "es2020" } }
```

Q18: What are the roles of `rootDir` and `outDir` in `tsconfig.json` ?
A18:

- **`rootDir`** : Specifies the root directory for `.ts` files (e.g., `"src"`).
- **`outDir`** : Defines the output directory for compiled `.js` files (e.g., `"dist"`). Example:

```
{ "compilerOptions": { "rootDir": "src", "outDir": "dist" } }
```

Q19: How does the `noImplicitAny` option affect compilation?
A19:

- **`true`** : Errors on implicit `any` types (e.g., `const greet = (name) => ...` fails).
- **`false`** : Allows implicit `any` without errors. Example:

```
{ "compilerOptions": { "noImplicitAny": true } }
```

Q20: What does the `removeComments` option do?
A20:

- **`true`** : Strips comments from compiled JavaScript.
- **`false`** : Retains comments. Example:

```
{ "compilerOptions": { "removeComments": true } }
```

Interfaces

Q21: How is an interface used to type an object in TypeScript?
A21: An interface defines an object’s shape. Example:

```
interface User {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
}
const user: User = {
  firstName: "harkirat",
  lastName: "singh",
  email: "email@gmail.com",
  age: 21,
};
```

Q22: How does the `isLegal` function use an interface in Assignment 1?

A22: The `isLegal` function:

- Takes a `User` interface parameter and checks if `age > 18`. Example:

```
interface User {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
}
function isLegal(user: User): boolean {
  return user.age > 18;
}
```

Q23: How is an interface used in the `Todo` React component in Assignment 2?

A23: The `Todo` component:

- Uses `TodoType` for the `todo` prop and `TodoInput` for the prop structure. Example:

```
interface TodoType {
  title: string;
  description: string;
  done: boolean;
}
interface TodoInput {
  todo: TodoType;
}
function Todo({ todo }: TodoInput): JSX.Element {
  return (
    <div>
      <h1>{todo.title}</h1>
      <h2>{todo.description}</h2>
    </div>
  );
}
```

Q24: How can a class implement an interface in TypeScript?

A24: A class uses `implements` to adhere to an interface. Example:

```
interface Person {
  name: string;
  age: number;
  greet(phrase: string): void;
}
class Employee implements Person {
  name: string;
  age: number;
  constructor(n: string, a: number) {
    this.name = n;
    this.age = a;
  }
  greet(phrase: string) {
    console.log(`${phrase} ${this.name}`);
  }
}
```

Types

Q25: How does a type define an object structure, and what is an example?

A25: A type uses `type` to define an object's structure. Example:

```
type User = {
  firstName: string;
  lastName: string;
  age: number;
};
const user: User = {
  firstName: "harkirat",
  lastName: "singh",
  age: 21,
};
```

Q26: What is a union type, and how is it used in the `printId` example?

A26: A union type allows a value to be one of several types. Example:

```
type StringOrNumber = string | number;
function printId(id: StringOrNumber) {
  console.log(`ID: ${id}`);
}
printId(101); // ID: 101
printId("202"); // ID: 202
```

Q27: What is an intersection type, and how is it demonstrated in the `TeamLead` example?

A27: An intersection type combines multiple types using `&`. Example:

```
type Employee = { name: string; startDate: Date; };
type Manager = { name: string; department: string; };
type TeamLead = Employee & Manager;
const teamLead: TeamLead = {
  name: "harkirat",
  startDate: new Date(),
  department: "Software Developer",
};
```

Interfaces vs Types

Q28: What are the major differences between interfaces and types in TypeScript?
A28: Major differences:

1. **Declaration Syntax**:
   - **Type**: Uses `type` (e.g., `type User = {...}`).
   - **Interface**: Uses `interface` (e.g., `interface User {...}`).
2. **Extension and Merging**:
   - **Type**: Supports extension but doesn't merge; redefinition overrides.
   - **Interface**: Extends with `extends` and merges same-name declarations.
3. **Declaration vs. Implementation**:
   - **Type**: Represents any type (primitives, unions, etc.).
   - **Interface**: Focuses on object shapes and class contracts.

Q29: When should you use types vs interfaces?
A29:

- **Use Types**: For unions, intersections, primitives, or advanced type features (e.g., conditional types).
- **Use Interfaces**: For object shapes, class contracts, or when merging/extending is needed.

Q30: Provide an example contrasting a type and an interface from the content.
A30:

- **Type**:

```
type StringOrNumber = string | number;
function printId(id: StringOrNumber) {
  console.log(`ID: ${id}`);
}
printId(101);
```

- **Interface**:

```
interface Employee {
  name: string;
  startDate: Date;
}
const emp: Employee = { name: "Harkirat", startDate: new Date() };
```

These questions and answers encapsulate the core concepts from "Week 9.2: Introduction to TypeScript," providing a solid foundation in TypeScript’s purpose, execution, basic types, configuration, and the distinctions between interfaces and types, along with practical examples.

Below is a set of questions and answers derived directly from the content provided for "Week 12.2: Advanced TypeScript APIs." These questions focus on advanced TypeScript utility types (`Pick`, `Partial`, `Readonly`, `Record`, `Exclude`, and `Map`) and type inference in Zod, ensuring a comprehensive understanding of these concepts and their practical applications.

Prerequisites

Q1: What are the key prerequisites for understanding advanced TypeScript APIs?
A1: You should:

1. **Understand Basic TypeScript Classes**: Know how to define classes, constructors, properties, methods, and inheritance.
2. **Understand Interfaces and Types**: Be familiar with defining and using interfaces and types for object structures and function parameters.
3. **Experience with TypeScript in Node.js**: Have experience setting up and running a TypeScript Node.js project.

Q2: What does the example code snippet test in terms of understanding?
A2: The snippet:

```
interface User {
  name: string;
  age: number;
}
function sumOfAge(user1: User, user2: User) {
  return user1.age + user2.age;
}
const result = sumOfAge({ name: "harkirat", age: 20 }, { name: "raman", age: 21 });
console.log(result); // Output: 41
```

Tests understanding of:

- **Interface `User`**: Defines an object structure.
- **Function `sumOfAge`**: Uses typed parameters and returns a number.
- **Usage**: Passing objects that conform to the interface.

Q3: How do you set up a TypeScript project locally?
A3: Steps:

1. Run `npx tsc --init` to create `tsconfig.json`.
2. Edit `tsconfig.json`:

```
{
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

- `rootDir`: Source directory for `.ts` files.
- `outDir`: Output directory for compiled `.js` files.

1] Pick

Q4: What does the `Pick` utility type do in TypeScript?
A4: `Pick<Type, Keys>` constructs a new type by selecting a subset of properties (`Keys`) from

an existing type (`Type`). Syntax: `Pick<User, 'name' | 'email'>` .

Q5: How is `Pick` used in an example with a `User` interface?
A5: Example:

```
interface User {
  id: number;
  name: string;
  email: string;
  createdAt: Date;
}
type UserProfile = Pick<User, 'name' | 'email'>;
const displayUserProfile = (user: UserProfile) => {
  console.log(`Name: ${user.name}, Email: ${user.email}`);
};
```

- `UserProfile` includes only `name` and `email` from `User` .

Q6: What are the benefits of using `Pick` ?
A6: Benefits:

1. **Enhanced Type Safety**: Limits properties to those needed, reducing errors.
2. **Code Readability**: Clarifies intent with specific types.
3. **Reduced Redundancy**: Reuses existing types without manual duplication.

2] Partial

Q7: What is the purpose of the `Partial` utility type?
A7: `Partial<Type>` makes all properties of `Type` optional, useful for partial updates. Syntax: `Partial<User>` .

Q8: How does `Partial` work with `Pick` in an update function?
A8: Example:

```
interface User {
  id: string;
  name: string;
  age: string;
  email: string;
  password: string;
}
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>;
type UpdatePropsOptional = Partial<UpdateProps>;
function updateUser(updatedProps: UpdatePropsOptional) {
  // Update logic
}
updateUser({ name: "Alice" }); // Valid
updateUser({ age: "30", email: "alice@example.com" }); // Valid
```

- `UpdatePropsOptional` makes `name`, `age`, and `email` optional.

Q9: What advantages does `Partial` provide?

A9: Advantages:

1. **Flexibility in Updates**: Allows updating only some properties.
2. **Type Safety**: Maintains type checking for provided properties.
3. **Code Simplicity**: Simplifies function signatures for partial updates.

3] Readonly

Q10: What does the `Readonly` utility type do?

A10: `Readonly<Type>` makes all properties of `Type` read-only, preventing reassignment after creation. Syntax: `Readonly<Config>`.

Q11: How is `Readonly` applied to a configuration object?

A11: Example:

```
interface Config {
  endpoint: string;
  apiKey: string;
}
const config: Readonly<Config> = {
  endpoint: 'https://api.example.com',
  apiKey: 'abcdef123456',
};
// config.apiKey = 'newkey'; // Error: Cannot assign to 'apiKey'
```

- `config` cannot be modified after initialization.

Q12: What are the benefits and limitations of `Readonly`?
A12:

- **Benefits**:
  i. **Immutability**: Prevents changes post-creation.
  ii. **Compile-Time Checking**: Catches errors early.
  iii. **Clarity**: Signals intent for immutability.
- **Limitation**: Enforced only at compile-time, not runtime, as JavaScript lacks native immutability.

4] Record & Map

Q13: What is the `Record` utility type, and how is it used?

A13: `Record<K, T>` creates a type with keys of type `K` and values of type `T`. Example:

```
interface User {
  id: string;
  name: string;
}
type Users = Record<string, User>;
const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};
```

- `Users` maps string keys to `User` objects.

Q14: How does the `Map` object work in TypeScript?

A14: `Map` stores key-value pairs with any key type and maintains insertion order. Example:

```
interface User {
  id: string;
  name: string;
}
const usersMap = new Map<string, User>();
usersMap.set('abc123', { id: 'abc123', name: 'John Doe' });
console.log(usersMap.get('abc123')); // { id: 'abc123', name: 'John Doe' }
```

Q15: When should you use `Record` versus `Map`?

A15:

- **Use `Record`**: For objects with fixed value shapes and string keys.
- **Use `Map`**: For flexible keys, order preservation, or frequent additions/removals.

5] Exclude

Q16: What does the `Exclude` utility type do?

A16: `Exclude<T, U>` removes members of `U` from the union type `T`, creating a subset type. Syntax: `Exclude<Event, 'scroll'>`.

Q17: How is `Exclude` used to restrict event types?

A17: Example:

```
type Event = 'click' | 'scroll' | 'mousemove';
type ExcludeEvent = Exclude<Event, 'scroll'>; // 'click' | 'mousemove'
const handleEvent = (event: ExcludeEvent) => {
  console.log(`Handling event: ${event}`);
};
handleEvent('click'); // OK
// handleEvent('scroll'); // Error
```

- `ExcludeEvent` excludes `'scroll'` from `Event`.

Q18: What are the benefits of using `Exclude`?

A18: Benefits:

1. **Type Safety**: Prevents unwanted types.
2. **Code Readability**: Clarifies type restrictions.
3. **Utility**: Simplifies type refinement.

6] Type Inference in Zod

Q19: How does type inference work in Zod?

A19: Zod infers TypeScript types from schema definitions, aligning runtime validation with compile-time safety. Example:

```
const userProfileSchema = z.object({
  name: z.string().min(1),
  email: z.string().email(),
  age: z.number().min(18).optional(),
});
const result = userProfileSchema.safeParse(req.body);
if (result.success) {
  const updateBody = result.data; // Inferred type: { name: string; email: string; age?: number }
}
```

Q20: What is an example of using Zod in an Express app?

A20: Example:

```
import { z } from 'zod';
import express from "express";
const app = express();
app.use(express.json());
const userProfileSchema = z.object({
  name: z.string().min(1),
  email: z.string().email(),
  age: z.number().min(18).optional(),
});
app.put("/user", (req, res) => {
  const result = userProfileSchema.safeParse(req.body);
  if (!result.success) {
    res.status(400).json({ error: result.error });
    return;
  }
  const updateBody = result.data; // Type inferred
  res.json({ message: "User updated", updateBody });
});
app.listen(3000);
```

Q21: What are the benefits of type inference in Zod?
A21: Benefits:

1. **Reduced Boilerplate**: Eliminates manual type definitions.
2. **Type Safety**: Ensures schema-validated data is typed correctly.
3. **Developer Productivity**: Simplifies validation and type management.

These questions and answers cover the core concepts from "Week 12.2: Advanced TypeScript APIs," including utility types (`Pick`, `Partial`, `Readonly`, `Record`, `Exclude`, `Map`) and Zod's type inference, providing a solid foundation for leveraging these features in TypeScript projects.