Basics

1. Introduction to TypeScript

- **What is TypeScript?**

  TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It adds optional static types to the language, enabling developers to catch errors early and write more maintainable code.

- **Setting up the development environment**

  To set up TypeScript, you need to install Node.js and npm (Node Package Manager). Then, install TypeScript globally using the command:

  ```sh
  npm install -g typescript
  ```

  You can then create a TypeScript file (`.ts`) and compile it to JavaScript using the TypeScript compiler (`tsc`):

  ```sh
  tsc filename.ts
  ```

- **Basic syntax and types**

  TypeScript syntax is similar to JavaScript, with added type annotations:

  ```typescript
  let isDone: boolean = false;
  let decimal: number = 6;
  let color: string = "blue";
  let list: number[] = [1, 2, 3];
  let tuple: [string, number];
  tuple = ["hello", 10]; // OK
  ```

2. Type Annotations

- **Primitive types: string, number, boolean, etc.**

  ```typescript
  let isDone: boolean = true;
  let decimal: number = 6;
  let color: string = "red";
  ```

- **Array and tuple types**
  ```typescript
  let list: number[] = [1, 2, 3];
  let tuple: [string, number];
  tuple = ["hello", 10];
  ```

- **Enum types**
  ```typescript
  enum Color {Red, Green, Blue}
  let c: Color = Color.Green;
  ```

- **Any and unknown types**
  ```typescript
  let notSure: any = 4;
  notSure = "maybe a string instead";
  notSure = false; // okay, definitely a boolean

  let uncertain: unknown = 4;
  uncertain = "maybe a string instead";
  ```

  uncertain = false; // okay, definitely a boolean

- **Type inference**
  TypeScript can infer types based on the assigned values:
  ```typescript
  let x = 3; // x is inferred to be a number
  ```

Advanced Types

3. Interfaces
- **Defining interfaces**
  ```typescript
  interface LabelledValue {
    label: string;
  }

  function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
  ```

```
  }

  let myObj = {size: 10, label: "Size 10 Object"};
  printLabel(myObj);
```

- **Optional properties**

  ```typescript
  interface SquareConfig {
    color?: string;
    width?: number;
  }

  function createSquare(config: SquareConfig): {color: string; area: number} {
    let newSquare = {color: "white", area: 100};
    if (config.color) {
      newSquare.color = config.color;
    }
    if (config.width) {
      newSquare.area = config.width * config.width;
  ```

```typescript
    }
    return newSquare;
  }

  let mySquare = createSquare({color: "black"});
  ```

- **Readonly properties**
  ```typescript
  interface Point {
    readonly x: number;
    readonly y: number;
  }

  let p1: Point = { x: 10, y: 20 };
  // p1.x = 5; // Error: Cannot assign to 'x' because it is a read-only property.
  ```

- **Function types**
  ```typescript
  ```

```
  interface SearchFunc {
    (source: string, subString: string): boolean;
  }

  let mySearch: SearchFunc;
  mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    return result > -1;
  }
  ```

- **Indexable types**
  ```typescript
  interface StringArray {
    [index: number]: string;
  }

  let myArray: StringArray;
  myArray = ["Bob", "Fred"];
  ```

```
let myStr: string = myArray[0];
```

4. Classes

- **Class basics**

  ```typescript
  class Greeter {
    greeting: string;
    constructor(message: string) {
      this.greeting = message;
    }
    greet() {
      return "Hello, " + this.greeting;
    }
  }

  let greeter = new Greeter("world");
  ```

- **Constructors**

  ```typescript
  ```

  class Animal {
    name: string;
    constructor(name: string) { this.name = name; }
    move(distanceInMeters: number = 0) {
      console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
  }
  ~~~

- **Inheritance and polymorphism**
  ```typescript
  class Dog extends Animal {
    bark() {
      console.log('Woof! Woof!');
    }
  }

  let dog = new Dog("Rex");
  dog.bark();
  dog.move(10);
  ~~~

  dog.bark();
  ~~~

- **Access modifiers (public, private, protected)**
  ```typescript
  class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
  }

  class Rhino extends Animal {
    constructor() { super("Rhino"); }
  }

  class Employee {
    private name: string;
    constructor(theName: string) { this.name = theName; }
  }
  ~~~

```
  let animal = new Animal("Goat");
  let rhino = new Rhino();
  let employee = new Employee("Bob");

  animal = rhino;
  // animal = employee; // Error: 'Animal' and 'Employee' are not compatible
```

- **Abstract classes**
  ```typescript
  abstract class Department {
    constructor(public name: string) {}

    printName(): void {
      console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // Must be implemented in derived classes
  }
  ```

```
  class AccountingDepartment extends Department {
    constructor() {
      super("Accounting and Auditing"); // Constructors in derived classes must call super()
    }

    printMeeting(): void {
      console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
      console.log("Generating accounting reports...");
    }
  }

  let department: Department; // OK to create a reference to an abstract type
  // department = new Department(); // Error: Cannot create an instance of an abstract class
```

  department = new AccountingDepartment(); // OK to create and assign a non-abstract subclass
  department.printName();
  department.printMeeting();
  // department.generateReports(); // Error: Method doesn't exist on declared abstract type

5. Functions
- **Function types and signatures**
  ```typescript
  function add(x: number, y: number): number {
    return x + y;
  }

  let myAdd = function(x: number, y: number): number { return x + y; };
  ```

- **Optional and default parameters**
  ```typescript
  ```

```
  function buildName(firstName: string, lastName?: string) {
    if (lastName)
      return firstName + " " + lastName;
    else
      return firstName;
  }

  let result1 = buildName("Bob");  // Works correctly now
  let result2 = buildName("Bob", "Adams", "Sr.");  // Error, too many parameters
  let result3 = buildName("Bob", "Adams");  // Ah, just right
  ```

- **Rest parameters**
  ```typescript
  function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
  }
  ```

~~~
  let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
  ```


- **Overloads**
  ```typescript
  function pickCard(x: {suit: string; card: number; }[]): number;
  function pickCard(x: number): {suit: string; card: number; };
  function pickCard(x): any {
    if (typeof x == "object") {
      return Math.floor(Math.random() * x.length);
    } else if (typeof x == "number") {
      return { suit: "hearts", card: x % 13 };
    }
  }

  let myDeck = [
    { suit: "diamonds", card: 2 },
~~~

```
    { suit: "spades", card: 10 },
    { suit: "hearts", card: 4 },
  ];
  let pickedCard1 = myDeck[pickCard(myDeck)];
  console.log("card: " + pickedCard1.card + " of " + pickedCard1.suit);

  let pickedCard2 = pickCard(15);
  console.log("card: " + pickedCard2.card + " of " + pickedCard2.suit);
  ```

Type Features

6. Generics

- **Generic functions**

  ```typescript
  function identity<T>(arg: T): T {
    return arg;
  }
  ```

  let output = identity<string>("myString"); // type of output will be 'string'

- **Generic classes**
  ```typescript
  class GenericNumber<T> {
    zeroValue: T;
    add

: (x: T, y: T) => T;
  }

  let myGenericNumber = new GenericNumber<number>();
  myGenericNumber.zeroValue = 0;
  myGenericNumber.add = function(x, y) { return x + y; };
  ```

- **Generic interfaces**

```typescript
  interface GenericIdentityFn<T> {
    (arg: T): T;
  }

  function identity<T>(arg: T): T {
    return arg;
  }

  let myIdentity: GenericIdentityFn<number> = identity;
  ```

- **Constraints**
  ```typescript
  interface Lengthwise {
    length: number;
  }

  function loggingIdentity<T extends Lengthwise>(arg: T): T {
  ```

    console.log(arg.length); // Now we know it has a .length property, so no more error
    return arg;
  }

  loggingIdentity({length: 10, value: 3});

7. Modules
- **Exporting and importing modules**
  ```typescript
  // math.ts
  export function add(x: number, y: number): number {
    return x + y;
  }

  // app.ts
  import { add } from "./math";
  console.log(add(5, 3));
  ```

- **Default exports**
  ```typescript
  // math.ts
  export default function add(x: number, y: number): number {
    return x + y;
  }

  // app.ts
  import add from "./math";
  console.log(add(5, 3));
  ```

- **Namespaces**
  ```typescript
  namespace Validation {
    export interface StringValidator {
      isAcceptable(s: string): boolean;
    }
  ```

```
  const lettersRegexp = /^[A-Za-z]+$/;
  const numberRegexp = /^[0-9]+$/;

  export class LettersOnlyValidator implements
StringValidator {
   isAcceptable(s: string) {
    return lettersRegexp.test(s);
   }
  }

  export class ZipCodeValidator implements
StringValidator {
   isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
   }
  }
 }

 let strings = ["Hello", "98052", "101"];
 let validators: { [s: string]:
Validation.StringValidator; } = {};
```

```
  validators["ZIP code"] = new Validation.ZipCodeValidator();
  validators["Letters only"] = new Validation.LettersOnlyValidator();
  for (let s of strings) {
    for (let name in validators) {
      let isMatch = validators[name].isAcceptable(s);
      console.log(`"${s}" - ${isMatch ? "matches" : "does not match"} ${name}`);
    }
  }
  ```

8. Type Assertions

- **Casting types**

  ```typescript
  let someValue: any = "this is a string";
  let strLength: number = (<string>someValue).length;
  ```

- **Non-null assertions**

```typescript
  let s: string | null = "hello";
  s!.toUpperCase(); // OK
  ```

9. Utility Types
- **Partial, Readonly, Pick, Omit, etc.**
  ```typescript
  interface Todo {
    title: string;
    description: string;
  }

  function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
    return { ...todo, ...fieldsToUpdate };
  }

  const todo1 = {
    title: "organize desk",
    description: "clear clutter",
  ```

```
};

const todo2 = updateTodo(todo1, {
  description: "throw out trash",
});

const readOnlyTodo: Readonly<Todo> = {
  title: "Read-only title",
  description: "Read-only description",
};

type TodoPreview = Pick<Todo, "title">;

const todoPreview: TodoPreview = {
  title: "Only title",
};

type TodoOmit = Omit<Todo, "description">;

const todoOmit: TodoOmit = {
  title: "Only title",
```

 };
 ~~~

Advanced Topics

10. Decorators
- **Class decorators**
 ```typescript
 function sealed(constructor: Function) {
   Object.seal(constructor);
   Object.seal(constructor.prototype);
 }

 @sealed
 class Greeter {
   greeting: string;
   constructor(message: string) {
     this.greeting = message;
   }
   greet() {
     return "Hello, " + this.greeting;
 ~~~

    }
  }
  ~~~

- **Method decorators**
  ```typescript
  function enumerable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
      descriptor.enumerable = value;
    };
  }

  class Greeter {
    greeting: string;
    constructor(message: string) {
      this.greeting = message;
    }

    @enumerable(false)
    greet() {
  ~~~

```
    return "Hello, " + this.greeting;
  }
}
```

- **Property decorators**

  ```typescript
  function format(prefix: string) {
    return function (target: any, propertyKey: string) {
      let _val = target[propertyKey];
      const getter = () => `${prefix} ${_val}`;
      const setter = (newVal: string) => { _val = newVal; };

      Object.defineProperty(target, propertyKey, {
        get: getter,
        set: setter,
        enumerable: true,
        configurable: true
      });
    };
  ```

```
  }

  class Greeter {
    @format("Hello")
    greeting: string;

    constructor(message: string) {
      this.greeting = message;
    }
  }

  let greeter = new Greeter("world");
  console.log(greeter.greeting); // "Hello world"
```

- **Parameter decorators**

```typescript
  function logParameter(target: any, propertyKey: string, parameterIndex: number) {
    const existingMetadata = Reflect.getOwnMetadata("logParameter", target, propertyKey) || [];
    existingMetadata.push(parameterIndex);
    Reflect.defineMetadata("logParameter", existingMetadata, target, propertyKey);
  }

  class Greeter {
    greeting: string;
    constructor(message: string) {
      this.greeting = message;
    }

    greet(@logParameter message: string) {
      console.log(`Greeting: ${message}`);
    }
  }
```

11. Mixins

- **Creating mixins**
  ```typescript
  class Disposable {
    isDisposed: boolean;
    dispose() {
      this.isDisposed = true;
    }
  }

  class Activatable {
    isActive: boolean;
    activate() {
      this.isActive = true;
    }
    deactivate() {
      this.isActive = false;
    }
  }

  class SmartObject implements Disposable, Activatable {
  ```

```
  constructor() {
   setInterval(() => console.log(this.isActive + " : " +
this.isDisposed), 500);
  }

  interact() {
   this.activate();
  }

  // Disposable
  isDisposed: boolean = false;
  dispose: () => void;

  // Activatable
  isActive: boolean = false;
  activate: () => void;
  deactivate: () => void;
 }

 applyMixins(SmartObject, [Disposable,
Activatable]);
```

```
  let smartObj = new SmartObject();
  setTimeout(() => smartObj.interact(), 1000);

  function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
      Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
        derivedCtor.prototype[name] = baseCtor.prototype[name];
      });
    });
  }
```

- **Applying mixins**
  Mixin functions can be applied to extend functionality dynamically.

12. Namespaces and Modules

- **Internal and external modules**

  ```typescript
  namespace Shapes {
    export namespace Polygons {
      export class Triangle { }
      export class Square { }
    }
  }

  import polygons = Shapes.Polygons;
  let sq = new polygons.Square(); // Same as "new Shapes.Polygons.Square()"
  ```

- **Namespaces vs. modules**

  Namespaces are used to organize code within a file or across multiple files, while modules are used to organize code across files and packages.

13. Type Guards

- **Typeof, instanceof, and custom type guards**

```typescript
  function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
      return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
      return padding + value;
    }
    throw new Error(`Expected string or number, got '${typeof padding}'.`);
  }

  function isNumber(x: any): x is number {
    return typeof x === "number";
  }

  function isString(x: any): x is string {
    return typeof x === "string";
  }
```

14. Advanced Types and Concepts
- **Union and intersection types**
  ```typescript
  function merge<T, U>(obj1: T, obj2: U): T & U {
    let result = <T & U>{};
    for (let id in obj1) {
      (<any>result)[id] = (<any>obj1)[id];
    }
    for (let id in obj2) {
      if (!result.hasOwnProperty(id)) {
        (<any>result)[id] = (<any>obj2)[id];
      }
    }
    return

 result;
  }

  let merged = merge({name: "John"}, {age: 25});
  console.log(merged.name); // John
  ```

  console.log(merged.age); // 25

- **Type aliases**
  ```typescript
  type Name = string;
  type NameResolver = () => string;
  type NameOrResolver = Name | NameResolver;

  function getName(n: NameOrResolver): Name {
    if (typeof n === "string") {
      return n;
    } else {
      return n();
    }
  }
  ```

- **Conditional types**
  ```typescript
  ```

```
  type MessageOf<T> = T extends { message: unknown } ? T["message"] : never;

  interface Email {
    message: string;
  }

  type EmailMessageContents = MessageOf<Email>; // string
  ```

- **Discriminated unions**
  ```typescript
  interface Bird {
    kind: "bird";
    fly(): void;
  }

  interface Fish {
    kind: "fish";
    swim(): void;
  ```

```
}

type Pet = Bird | Fish;

function getSmallPet(): Pet {
  return Math.random() > 0.5 ? { kind: "bird", fly: () => {} } : { kind: "fish", swim: () => {} };
}

let pet = getSmallPet();

if (pet.kind === "bird") {
  pet.fly();
} else {
  pet.swim();
}
```

Integration and Tools

15. Tooling and Frameworks

- **Setting up TypeScript with popular frameworks (React, Angular, Node.js)**

  TypeScript can be used with various frameworks. For React:

```sh
npx create-react-app my-app --template typescript
```

  For Angular:

```sh
ng new my-app
```

  For Node.js:

```sh
npm init -y
npm install typescript @types/node
tsc --init
```

- **Linters and formatters**

  Use TSLint or ESLint with TypeScript to enforce coding standards:

```sh
  npm install eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin --save-dev
  ```

- **Debugging TypeScript**

  TypeScript can be debugged using tools like VS Code, which provides built-in support for TypeScript debugging.

- **Testing TypeScript code**

  Use testing frameworks like Jest or Mocha with TypeScript:

  ```sh
  npm install --save-dev jest @types/jest ts-jest
  ```

16. Configuration and Compilation

- **tsconfig.json configuration**

  The `tsconfig.json` file is used to configure the TypeScript compiler options:

  ```json
  ```

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true
  }
}
```

- **Compiler options**

  The TypeScript compiler supports various options to control the compilation process. Some common options include:

  - `target`: Specifies the output JavaScript version.
  - `module`: Specifies the module system to use.
  - `strict`: Enables all strict type-checking options.

- **Integrating with build tools (Webpack, Gulp, etc.)**

  TypeScript can be integrated with build tools like Webpack:

```sh
npm install --save-dev typescript ts-loader webpack webpack-cli
```

Configure Webpack to use TypeScript:

```javascript
module.exports = {
  entry: './src/index.ts',
  module: {
    rules: [
      {
        test: /\.ts$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js']
  },
  output: {
```

      filename: 'bundle.js',
      path: __dirname + '/dist'
    }
  };
  ~~~

17. Migration to TypeScript

- **Strategies for migrating existing JavaScript codebases to TypeScript**

  Migrate JavaScript codebases to TypeScript incrementally by renaming `.js` files to `.ts` and gradually adding type annotations.

- **Incremental adoption patterns**

  Use the `allowJs` and `checkJs` compiler options to enable TypeScript to check JavaScript files:

  ```json
  {
    "compilerOptions": {
      "allowJs": true,
      "checkJs": true
    }
  ~~~

 }
 ```

Best Practices

18. Best Practices

- **Code organization and modularity**
  Organize code into modules and namespaces to improve maintainability and reusability.

- **Type safety and code quality**
  Use strict type-checking options and avoid using the `any` type to ensure type safety and high code quality.

- **Performance considerations**
  Minimize the use of heavy computation and optimize code to ensure good performance, especially in large TypeScript applications.
 ```