Chapter 6 – Advanced Object Types

1. Introduction
   1. One of the challenges of writing TypeScript is knowing how to apply types in every situation we’ll encounter within our code
   2. we’ll learn how to use types with object-oriented programming patterns, how to use types together to create combined types
2. Interfaces and Types
   1. there’s another way to define types with the interface keyword.
   2. The syntaxes for type and interface are slightly different, since interface does not require an equals sign (=) before the typed object

Example

|  |
| --- |
| type Mail = {  postagePrice: number;  address: string;  }    const catalog: Mail = ... |

Example 2

|  |
| --- |
| interface Mail {  postagePrice: number;  address: string;  }    const catalog: Mail = ... |

Exercise

|  |
| --- |
| // Write an interface here  interface Run {  miles: number  }  function updateRunGoal(run: Run) {  console.log(`  Miles left: ${50 - run.miles}  Percent of goal: ${(run.miles / 50) \* 100}% complete  `)  }  updateRunGoal({  miles: 5,  }) |

1. Interfaces and Classes
   1. The interface keyword in TypeScript is especially good for adding types to a class
      1. interface is constrained to typed objects
      2. class is a way to program with objects

Example

|  |
| --- |
| interface Robot {  identify: (id: number) => void;  }    class OneSeries implements Robot {  identify(id: number) {  console.log(`beep! I'm ${id.toFixed(2)}.`);  }    answerQuestion() {  console.log('42!');  }  } |

* 1. interface named Robot and a class named OneSeries. The implements keyword is then used to apply the type Robot to OneSeries
  2. implements and interface allow us to create types that match a variety of class patterns, which makes interface a good tool for use on object-oriented programs.

Exercise

|  |
| --- |
| // Write an interface here  interface Directory {  addFile: (name: string) => void  }  class DesktopDirectory implements Directory {  addFile(name: string) {  console.log(`Adding file: ${name}`);  }  showPreview(name: string) {  console.log(`Opening preview of file: ${name}`);  }  }  const Desktop = new DesktopDirectory();  Desktop.addFile('lesson-notes.txt');  Desktop.showPreview('lesson-notes.txt'); |

1. Deep Types
   1. As our programs grow and become more complex, we’ll need to add more methods and properties to our objects to accommodate more features.
   2. TypeScript allows us to infinitely nest objects so that we can describe data correctly.
      1. interface is constrained to typed objects
      2. class is a way to program with objects

Example

|  |
| --- |
| interface Robot {  about: {  general: {  id: number;  name: string;  };  };  getRobotId: () => string;  }  class OneSeries implements Robot {  about;    constructor(props: { general: { id: number; name: string; } }) {  this.about = props;  }    getRobotId() {  return `ID: ${this.about.general.id}`;  }  } |

Exercise

|  |
| --- |
| interface Directory {  addFile: (name: string) => void;  // Define a config type member here  config: {  default: {  encoding: string,  permissions: string  }  }  }  class DesktopDirectory implements Directory {  config = {  default: {  encoding: 'utf-8',  permissions: 'drw-rw-rw-',  }  }  addFile(name: string) {  console.log(`Adding file: ${name}`);  }  showPreview(name: string) {  console.log(`Opening preview of file: ${name}`);  }  }  const Desktop = new DesktopDirectory();  console.log(Desktop.config); |

1. Composed Types
   1. As our data gets nested deeper, we’ll start to have typed objects that become unwieldy to write and read. Take the following type: