Chapter 4 – Union Types

1. Introduction

A picture containing shape

Description automatically generated

* 1. any type is very unspecific.
  2. string. This type is very specific
  3. Union Types strikes balance between the two
     1. Allows to combine multiple specific types

1. Defining Unions
   1. Unions allow us to define multiple allowed *type members*
   2. Separating each type member with a vertical line character **|**

Example

|  |
| --- |
| let ID: string | number;    // number  ID = 1;    // or string  ID = '001';    console.log(`The ID is ${ID}.`); |

Exercise

|  |
| --- |
| function printNumsAndStrings(statement: string | number) {  console.log(`ℹ️ LOG:: ${statement}`);  }  printNumsAndStrings('hello!');  printNumsAndStrings(4); |

1. Type Narrowing
   1. Type with unions give flexibility, but there is more to consider
      1. We may want to perform different logic that does one thing for string and another for numbers 🡪 done using **Type Guard**

Example (Type Guard)

|  |
| --- |
| function formatValue(value: string | number) {  // Write your code here  }  formatValue('Hiya');  formatValue(42);  unction getMarginLeft(margin: string | number) {  // margin may be a string or number here    if (typeof margin === 'string') {  // margin must be a string here  return margin.toLowerCase();  }  } |

Exercise

|  |
| --- |
| function formatValue(value: string | number) {  // Write your code here  if (typeof value == "string") {  console.log(value.toLowerCase());  }  if (typeof value == "number") {  console.log(value.toFixed(2));  }  }  formatValue('Hiya');  formatValue(42); |

1. Inferred Union Return Types
   1. Typescript is able to infer types in many cases
   2. One of the useful case is return type of function
      1. If successful, return **Book** type
      2. If error, return **string** type

Exercises

|  |
| --- |
| function getBook() {  try {  return getBookFromServer();  } catch (error) {  return `Something went wrong: ${error}`;  }  } |

* 1. **Since TypeScript can infer the function’s return type, there’s no need for us to manually define it.**

Exercise

|  |
| --- |
| type User = {  id: number;  username: string;  };  function createUser() {  const randomChance = Math.random() >= 0.5;  if (randomChance) {  return { id: 1, username: 'nikko' };  } else {  return 'Could not create a user.';  }  }  let userData: User | string = createUser(); |

1. Unions and Arrays
   1. Unions are even more powerful if used with array
   2. To create a union that supports multiple types for an array’s values, wrap the union in parentheses (string | number), then use array notation []

Example

|  |
| --- |
| const dateNumber = new Date().getTime(); // returns a number  const dateString = new Date().toString(); // returns a string    const timesList: (string | number)[] = [dateNumber, dateString]; |

Exercise

|  |
| --- |
| function formatListings(listings: (number | string)[]) {  return listings.map((listing) => {  if (typeof listing === 'string') {  return listing.toUpperCase();  }  if (typeof listing === 'number') {  return `$${listing.toLocaleString()}`;  }  });  }  const result = formatListings([  '123 Main St',  226800,  '580 Broadway Apt 4a',  337900,  ]);  console.log(result); |

1. Common Key Value Pairs
2. TypeScript will only allow us to use the common methods and properties that all members of the union share

Example

|  |
| --- |
| const batteryStatus: boolean | number = false;    batteryStatus.toString(); // No TypeScript error  batteryStatus.toFixed(2); // TypeScript error |

1. Here, toString() exists for both type **Boolean** and **string**, but **toFixed()** doesn’t exist in both
2. This also applies to objects
   * 1. **isPettable** exists in both **Moose** and **Goose 🡪** No type error
     2. **hasHoofs** exists in only **Moose** 🡪 type error

Example 2

|  |
| --- |
| type Goose = {  isPettable: boolean;  hasFeathers: boolean;  canThwartAPicnic: boolean;  }    type Moose = {  isPettable: boolean;  hasHoofs: boolean;  }    const pettingZooAnimal: Goose | Moose = { isPettable: true };    console.log(pettingZooAnimal.isPettable); // No TypeScript error  console.log(pettingZooAnimal.hasHoofs); // TypeScript error |

Exercise

|  |
| --- |
| type Like = {  username: string;  displayName: string;  };  type Share = {  username: string;  displayName: string;  };  function getFriendNameFromEvent(event: Like | Share) {  return event.displayName || event.username;  }  const newEvent = {  username: 'vkrauss',  displayName: 'Veronica Krauss',  };  const friendName = getFriendNameFromEvent(newEvent);  console.log(`You have an update from ${friendName}.`); |

1. Union with Literal Types
   1. Literal type unions are useful when we want to create distinct states within a program
      1. Like enums

Exercise

|  |
| --- |
| type Color = 'green' | 'yellow' | 'red';  type Status = 'idle' | 'downloading' | 'complete';  function changeLight(color: Color) {  // ...  }  function downloadStatus(status: Status): void {  if (status == 'idle') {  console.log('Download');  }    if (status == 'downloading') {  console.log('Downloading...');  }  if (status == 'complete') {  console.log('Your download is complete!');  }  }  downloadStatus('idle');  downloadStatus('downloading');  downloadStatus('complete'); |

**Quiz**

1. Which call to carState() is valid?

**Graphical user interface

Description automatically generated with medium confidence**

1. **D**
2. What is the term for when TypeScript is able to infer a more specific type inside a program as the result of a type guard?

**Background pattern

Description automatically generated**

1. **A**
2. What is one situation where a union is preferred to typing a variable as any?