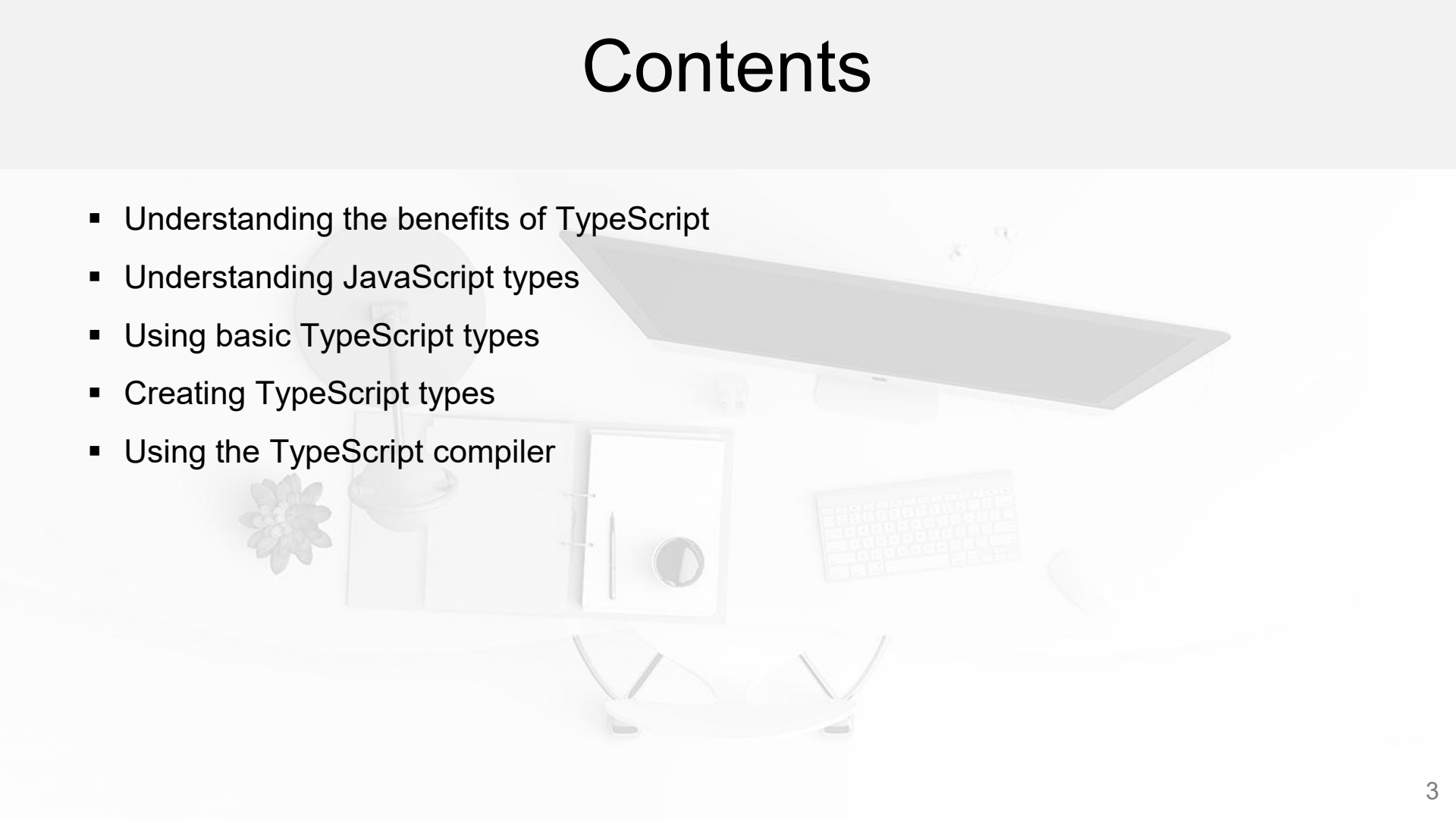
Advanced Web Programming

Ung Văn Giàu
Email: giau.ung@eiu.edu.vn

Introducing TypeScript

Contents

- Understanding the benefits of TypeScript
- Understanding JavaScript types
- Using basic TypeScript types
- Creating TypeScript types
- Using the TypeScript compiler



Technical requirements

- **Browser**: A modern browser such as Google Chrome.
- **TypeScript Playground**: https://www.typescriptlang.org/
- **CodeSandbox**: to explore JavaScript's type system (https://codesandbox.io/)
- **Visual Studio Code:** an editor to experience TypeScript's benefits and explore the TypeScript compiler.
- **Node.js** and **npm**: https://nodejs.org/en/download/.

1. Understanding the benefits of TypeScript

Understanding TypeScript

- TypeScript was first released in 2012 and is still being developed
- TypeScript is often referred to as a **superset or extension of JavaScript**
- TypeScript **can't be executed directly in a browser** – it must be transpiled into JavaScript first.
- TypeScript **adds a rich type system to JavaScript**.
  - It is generally used with frontend frameworks such as Angular, Vue, and React.
  - TypeScript can also be used to build a backend with Node.js.
- TypeScript **uses the type system to allow code editors to catch type errors** as developers write problematic code, and provide productivity features (code refactoring)

1. Understanding the benefits of TypeScript

Catching type errors early

- The type information helps the TypeScript compiler catch type errors.
- Carry out the following steps:
  - Open Visual Studio Code
  - Create a new file called **calculateTotalPrice.js**
  - Enter the following code:

```
function calculateTotalPriceJS(product, quantity,
discount) {
  const priceWithoutDiscount = product.price * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}
```

1. Understanding the benefits of TypeScript

A **.ts** file extension denotes a TypeScript file → a TypeScript compiler will perform type checking on this file.

Catching type errors early

- Carry out the following steps:
  - Create a copy of the file but with a **.ts** extension
  - Enter the following code:

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  quantity: number,
  discount: number
) {
  const priceWithoutDiscount = product.price * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}
```

1. Understanding the benefits of TypeScript

Improving developer experience and productivity with IntelliSense

- **IntelliSense** is a feature in code editors that **gives useful information** about elements of code and allows code to be **quickly completed** (list of properties available in an object).
- Carry out the following steps:
  - Open calculateTotalPrice.js > remove price.
  - With the cursor after the dot (.), click **Ctrl + spacebar** to opens Visual Studio Code's IntelliSense
  - Visual Studio Code can only guess the potential property name

1. Understanding the benefits of TypeScript

Improving developer experience and productivity with IntelliSense

- Carry out the following steps:
  - Open **calculateTotalPrice.ts**, remove **price** from product.price, and press **Ctrl + spacebar** to open IntelliSense again.
  - Visual Studio Code lists the correct properties.

1. Understanding the benefits of TypeScript

Recap

- TypeScript's **type-checking feature** helps catch problems earlier in the development process
- TypeScript enables code editors to **offer productivity features** such as IntelliSense
- These advantages provide significant benefits when working in larger codebases

2. Understanding JavaScript types

- Open the **CodeSandbox** and carry out the following steps:
  - Create a new plain JavaScript project
  - Open **index.js** > enter the following code:

```
let firstName = "Fred"
console.log("firstName", firstName, typeof firstName);
let score = 9
console.log("score", score, typeof score);
let date = new Date(2022, 10, 1);
console.log("date", date, typeof date);
```

  - Open the console and view the outputs

2. Understanding JavaScript types

- Open the CodeSandbox and carry out the following steps:
  - Add another couple of lines of code

```
score = "ten"
console.log("score", score, typeof score);
```

  - View the outputs in the console

2. Understanding JavaScript types

- **JavaScript only has a minimal set of types**, such as string, number, and boolean.

- JavaScript **allows a variable to change its type** → JavaScript engine won't throw an error if a variable is changed to a completely different type → impossible for code editors to catch type errors.

3. Using basic TypeScript types

Using type annotations

- TypeScript **type annotations** enable variables to be declared with specific types → It allow TypeScript to catch bugs where code uses the wrong type much earlier.
- The **type annotation comes after the variable declaration**. It starts **with a colon followed by the type** we want to assign to the variable
- Open the TypeScript Playground (https://www.typescriptlang.org/play) and carry out the following steps:
  - Remove any existing code in the left-hand pane and enter the following variable declaration: let unitPrice: number;
  - View the transpiled JavaScript on the right-hand side

3. Using basic TypeScript types

Using type annotations

- You can also **add type annotations to function parameters and a function's return value using the same syntax** as annotating a variable.
- Carry out the following steps:
  - Add a second line to the program: unitPrice = "Table"; (A type error being caught)
  - Enter the following function

```
function getTotal(
  unitPrice: number,
  quantity: number,
  discount: number
): number {
  const priceWithoutDiscount = unitPrice * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}
```

3. Using basic TypeScript types

Using type annotations

- Carry out the following steps:
  - Add another line of code to call getTotal with an incorrect type for quantity. Assign the result of the call to getTotal to a variable with an incorrect type:

```
let total: string = getTotal(500, "one", 0.1);
```

3. Using basic TypeScript types

Using type inference

- TypeScript's **type inference system** helps TypeScript **infers the type of a variable** when it is assigned a value from that value.
- Carry out the following steps:
  - Remove previous code and then add the following code: let flag = false;
  - Hover over the flag variable. A tooltip will appear showing the type that flag has been inferred to.
  - Add another line beneath this to incorrectly set flag to an invalid value: flag = "table";

3. Using basic TypeScript types

Using the Date type

- A Date type doesn't exist in JavaScript, but it existes in TypeScript.

  The TypeScript Date type is a representation of the **JavaScript Date object**.

- Carry out the following steps:
  - Remove any previous code and then add the following lines:

```
let today: Date;
today = new Date();
```

  - Refactor these two lines by useing type inference: let today = new Date();
  - Hover over *today* and checking the tooltip
  - Check IntelliSense is working by adding ***today.*** on a new line
  - Remove the line and add a slightly different line of code: today.addMonths(2);

3. Using basic TypeScript types

Using the any type

- TypeScript gives a variable with **no type** annotation and **no immediately assigned value** the ***any*** type.
- It is a way of opting **out of performing type checking** on a particular variable and is commonly **used for dynamic content or values from third-party libraries**.
- You need to **use any less often**
- Example:

  Enter *let flag;* in the TypeScript Playground and hover the mouser over the flag

3. Using basic TypeScript types

Using the unknown type

- ***unknown*** is a type we can use when we are unsure of the type but want to interact with it in a stronglytyped manner.
- Carry out the following steps:
  - Open the TypeScript Playground and enter the following:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data) => {
    console.log("firstName", data.firstName);
  });
```

  - Click on the Run option to execute the code

3. Using basic TypeScript types

Using the unknown type

- The **unknown** type contains nothing within its type. However, a variable's type can be widened.
- Carry out the following steps:
  - Give data the unknown type annotation:

```
fetch("https://swapi.dev/api/people/1")
 .then((response) => response.json())
 .then((data: unknown) => {
   console.log("firstName", data.firstName);
 });
```

  - A type error is now raised where firstName is referenced.

3. Using basic TypeScript types

Using the unknown type

- Carry out the following steps:
  - Change the firstName property to name

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    console.log("name", data.name);
  });
```

  - *name* is a valid property, but a type error is still occurring. This is because *data* is still unknown.

3. Using basic TypeScript types

Using the unknown type

- Carry out the following steps:
  - Change the code to widen the data type
  - **Notice** the return type of **isCharacter**, which is: character is { name: string }
  - This is a **type predicate**. TypeScript will narrow or widen the type of *character* to *{ name: string }* if the function returns ***true***.
  - Hover over the data variable, data starts off with the unknown type where it is assigned with a type annotation

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    if (isCharacter(data)) {
      console.log("name", data.name);
    }
  });

function isCharacter(
  character: any
): character is { name: string } {
  return "name" in character;
}
```

3. Using basic TypeScript types

Using the unknown type

- The unknown type is an excellent choice for data whose type you are unsure about.
- You can’t interact with unknown variables – the variable must be widened to a different type before any interaction.

3. Using basic TypeScript types

Using the void type

- The void type is used to represent a function's return type where the **function doesn't return a value**.
- If you use undefined as the return type, this raises a type error because a return type of undefined means that the function is expected to return a value (of type undefined)
- Example:

```
function logText(text: string) {
  console.log(text);
}
```

3. Using basic TypeScript types

Using the never type

- The ***never*** type represents something that will never occur and is typically **used to specify unreachable code areas**.
- **Template literals** are great when we need to merge static text with variables. Template literals are enclosed by **backticks (` `)** and can include a JavaScript expression in curly braces prefixed with a dollar sign (**${expression}**).
- Example:

```
function foreverTask(taskName: string): never {
  while (true) {
    console.log(`Doing ${taskName} over and over again
      ...`);
  }
}
```

3. Using basic TypeScript types

Using arrays

- Arrays are structures that TypeScript inherits from JavaScript.
- We add type annotations to arrays as usual, but with square brackets [] at the end to denote that this is an array type.
- Arrays are one of the most common types used to structure data.
- Examples:
  - Usual: const numbers: number[] = [];
  - The Array **generic type syntax** can be used: const numbers: Array<number> = [];
  - Inferred type: const numbers = [1, 2, 3];

3. Using basic TypeScript types

Recap

- TypeScript adds many useful types to JavaScripts types, such as Date, and is capable of representing arrays.
- TypeScript can infer a variable's type from its assigned value.
- No type checking occurs on variables with the **any** type, so this type should be avoided.
- The **unknown** type is a strongly-typed alternative to any, but unknown variables must be widened to be interacted with.
- **void** is a return type for a function that doesn't return a value.
- The **never** type can be used to mark unreachable areas of code.
- **Array** types can be defined using square brackets after the array item type.

4. Creating TypeScript types

Using object types

- An object type in TypeScript is represented a bit like a JavaScript object literal.
- Property types are specified.
- Properties in the object definitions can be separated by semicolons or commas, but using a semicolon is common practice.

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  ...
) {
  ...
}
```

- Example:

```
let table = {name: "Table", unitPrice: 450};
```

4. Creating TypeScript types

Using object types

- The **? symbol** can be **used in functions for optional parameters**. For example, myFunction(requiredParam: string, optionalParam?: string).
- Example:

```
const table: { name: string; unitPrice?: number } = {
  name: "Table",
};
```

4. Creating TypeScript types

Creating type aliases

- Example:

```
const table: { name: string; unitPrice?: number } = ...;
const chair: { name: string; unitPrice?: number } = ...;
```

4. Creating TypeScript types

Creating type aliases

- Type aliases allow **existing types to be composed together** and improve the readability and reusability of types.
- A type alias refers to another type, and **the syntax** is as follows:

type YourTypeAliasName = AnExistingType;

- Examples:
  - Create a type alias for the product object structure

```
type Product = { name: string; unitPrice?: number };
```

  - Assign two variables to this Product type

```
let table: Product = { name: "Table" };
let chair: Product = { name: "Chair", unitPrice: 40 };
```

4. Creating TypeScript types

Creating type aliases

- A type alias can **extend another object using the & symbol**.
- A type that extends another using the & symbol is referred to as an **intersection type.**
- Example:

```
type DiscountedProduct = Product & { discount: number };

let chairOnSale: DiscountedProduct = {
    name: "Chair on Sale",
    unitPrice: 30,
    discount: 5
};
```

4. Creating TypeScript types

Creating type aliases

- A type alias can also be used to represent a function.
- Example:

```
type Purchase = (quantity: number) => void;
```

  - Use the Purchase type to create a purchase function property in the Product type

```
type Purchase = (quantity: number) => void;
type Product = {
  name: string;
  unitPrice?: number;
  purchase: Purchase;
};
```

4. Creating TypeScript types

Creating type aliases

- A type alias can also be used to represent a function.
- Example:

```
type Purchase = (quantity: number) => void;
```

  - Add a purchase function property to the table variable declarations

```
let table: Product = {
  name: "Table",
  purchase: (quantity) =>
    console.log(`Purchased ${quantity} tables`),
};
table.purchase(4);
```

4. Creating TypeScript types

Creating interfaces

- Object types can be created using TypeScript's interface syntax.
- An interface is created with the **interface keyword**, followed by its **name**, followed by the bits that make up the interface in curly brackets:

```
interface Product {
  ...
}
```

- Example:

```
interface Product {
  name: string;
  unitPrice?: number;
}
```

4. Creating TypeScript types

Creating interfaces

- An interface can **extend another interface** using the **extends** keyword.

```
interface DiscountedProduct extends Product {
  discount: number;
}
```

- An interface can also be used to represent a function.

```
interface Purchase {(quantity: number): void}
```

4. Creating TypeScript types

When should I use a type alias instead of an interface and vice versa?

The capabilities of type aliases and interfaces for creating object types are very similar.

The simple answer is that it is down to preference for object types. Type aliases can create types that interfaces can’t, though, such as union types.

4. Creating TypeScript types

Creating classes

- A class is a standard JavaScript feature that acts as a template for creating an object.
- Properties and methods defined in the class are automatically included in objects created from the class.

4. Creating TypeScript types

Creating classes

- Example 1: assign initial values

```
class Product {
  name = "";
  unitPrice = 0;
}
```

- Example 2: add a constructor to the class

```
class Product {
  name;
  unitPrice;
  constructor(name: string, unitPrice: number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
}
```

4. Creating TypeScript types

Creating classes

- Example 3: The properties don’t need to be defined if the constructor parameters are marked as public.

```
class Product {
  constructor(public name: string, public unitPrice:
    number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
}
```

4. Creating TypeScript types

Creating classes

- Example 4: Type annotations can be added to method parameters and return values

```
class Product {
  constructor(public name: string, public unitPrice:
    number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
  getDiscountedPrice(discount: number): number {
    return this.unitPrice - discount;
  }
}
```

4. Creating TypeScript types

Creating classes

- Example 5: Create an instance of the class and output its discounted price to the console.

```
const table = new Product("Table", 45);
console.log(table.getDiscountedPrice(5));
```

4. Creating TypeScript types

Creating enumerations

- Enumerations allow us to **declare a meaningful set of friendly names** that a variable can be set to.
- Use the **enum** keyword, followed by the **name**, and then its possible **values** in curly braces.
- By **default**, enumerations are **zero-based numbers**.
- Example:
  - Create the enumeration

```
enum Level {
  Low,
  Medium,
  High
}
```

  - Create a level variable and assign it to the values

```
let level = Level.Low;
console.log(level);
```

4. Creating TypeScript types

Creating enumerations

- Instead of the default values, custom values can be explicitly defined against each enumeration item after the equals (=) symbol

```
enum Level {
  Low = 1,
  Medium = 2,
  High = 3
}
```

- Assign level to a number greater than 3: level = 10;

4. Creating TypeScript types

Creating enumerations

- Instead of using number enumeration values, let's try strings

```
enum Level {
  Low = "L",
  Medium = "M",
  High = "H"
}
```

- Add another line that assigns level to the following strings: level = "VH";

4. Creating TypeScript types

Creating enumerations

- Enumerations are a way of representing a range of values with user-friendly names.
- They are zero-based numbers by default and not as type-safe as we would like.
- We can make enumerations string-based.

4. Creating TypeScript types

Creating union types

- A union type is the mathematical **union of multiple other types** to create a new type.
- Union types can represent a range of values.
- Example:

```
type Level = "H" | "M" | "L";
```

4. Creating TypeScript types

Recap

- **Objects** and functions can be represented using **type** aliases or **interfaces**.
- The **?** symbol can specify that an object **property** or function **parameter** is **optional**.
- String-based enumerations are great for a specific set of strings. A string union type is the simplest approach if the strings are meaningful. If the strings aren't meaningful, then a string enumeration can be used to make them readable.

5. Using the TypeScript compiler

- Install the TypeScript compiler

  npm install -g typescript

  hoặc npm install --save-dev typescript

- Transpile TypeScript into JavaScript
  - Create a simple TS file (file_name.ts)
  - Open a terminal and type: **tsc** file_name.ts
  - Run JavaScript: **node** file_name.js


Q&A