MODULE 03

TYPESCRIPT TYPES AND OPERATORS

MODULE TOPICS

TypeScript Comments
Strong Typing with TypeScript
JavaScript Data Types
Coercion
Type Annotations
Types available in TypeScript
Type Assertions
TypeScript Operators
Type Aliases
Type Shapes
Inferred Typing

Type Compatibility

Duplicate Identifiers

TYPESCRIPT COMMENTS

TypeScript supports the traditional C based comments

```
// Single Line Comment
/* Multi
   Line
   Comment
*/
```

TYPESCRIPT COMMENTS

TypeScript Compiler can strip comments with removeComments compiler option

```
tsc complete.ts --removeComments
tsconfig.json
   { "compilerOptions": { "removeComments": true } }
```

TYPESCRIPT COMMENTS

TypeScript Triple-Slash Directives instruct the compiler to include additonal files in the compilation process

```
/// <reference lib="es2017.string" />
/// <amd-module name="NamedModule"/>
```

Triple-Slash Directives are only valid at the top of a TS file
If preceeded by a statement, they are treated as a single line commment

TYPESCRIPT COMMENTS

TSDoc is a Microsoft proposal to standardize doc comments for TypeScript

```
https://github.com/Microsoft/tsdoc
```

TYPESCRIPT COMMENTS

Similar to JSDoc without as much type annotations since TypeScript is strongly typed

```
/**
 * Returns the average of two numbers.
 *
 * @remarks
 * This method is part of the {@link core-library#Statistics | Statistics subsystem}.
 *
 * @param x - The first input number
 * @param y - The second input number
 * @returns The arithmetic mean of `x` and `y`
 *
 * @beta
 */
public getAverage(x: number, y: number): number { return (x + y) / 2.0; }
```

STRONG TYPING WITH TYPESCRIPT

JavaScript is a dynamically typed language
Variables, parameters, etc do not have a specific type
Types are inferred based on the current value

```
var variable1 = 10;             // number
variable1 = "10";               // string
variable1 = true;               // boolean
```

JAVASCRIPT DATA TYPES

boolean
number
string
object
null
undefined

New to ES6 / ES2015

symbol

COERCION

In JavaScript, type conversions are called "coercion"

"Explicit Coercion" is a forced conversion using functions such as Number(), String(), or .toString()

```
console.log( typeof "42" );                // Returns string
console.log( typeof Number( "42" ) );      // Returns number
```

"Impicit Coercion" are done automatically

```
console.log( "99.99" == 99.99 );     // Returns true, loosely equals
console.log( "99.99" === 99.99 );    // Returns false, strict equals
```

TYPESCRIPT DATA TYPES

boolean
number
string
object
null
undefined
symbol

Array
Tuple
enum

any / unknown

void

never

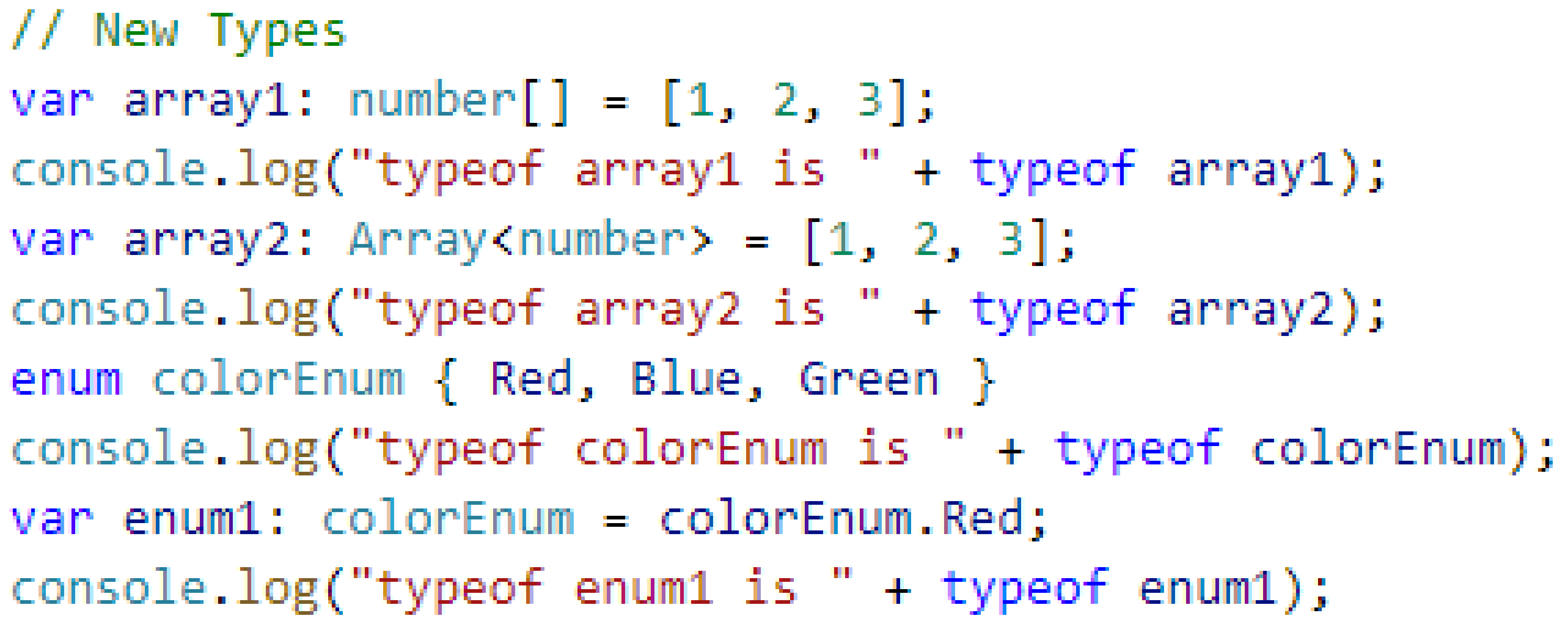
TYPE ANNOTATIONS

TypeScript's static data typing is implemented using Type Annotations

Variables, parameters, return types, etc can be typed when they are declared

```
var boolean1: boolean = true;
var number1: number = 5;
var string1: string = "string";
var object1: object = null;
var array1: Array<number> = [1, 2, 3];
```

WALKTHRU



```
// New Types
var array1: number[] = [1, 2, 3];
console.log("typeof array1 is " + typeof array1);
var array2: Array<number> = [1, 2, 3];
console.log("typeof array2 is " + typeof array2);
enum colorEnum { Red, Blue, Green }
console.log("typeof colorEnum is " + typeof colorEnum);
var enum1: colorEnum = colorEnum.Red;
console.log("typeof enum1 is " + typeof enum1);
```

TypeScript Types

TYPE ASSERTIONS

Casting in TypeScript is done with a "Type Assertion"
Two different syntaxes for a type assertions are available

as Syntax

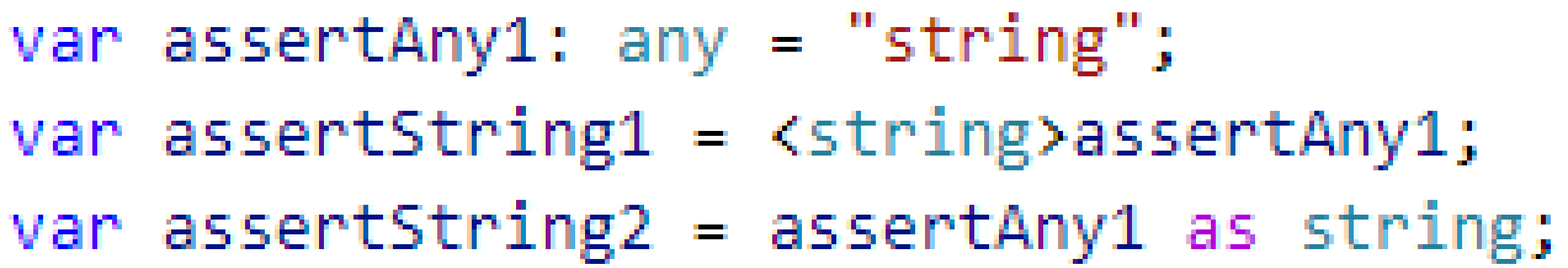
```
var assertAsString = assertAny1 as string;
```

Angle-bracket Syntax

```
var assertString1 = <string>assertAny1;
```

WALKTHRU

Type Assertions



```
var assertAny1: any = "string";
var assertString1 = <string>assertAny1;
var assertString2 = assertAny1 as string;
```

TYPESCRIPT OPERATORS

Operators	Types	Rules
Math - * / % + --	any, number, enum	Normal order of precedence
Add / Concat +	any, string, number, enum	Either is string, result is string
		Both or number or enum, result is number
		Either is any and no strings, result is any
Not !	all	Result is boolean
Comparison == !=	compatible types	Result is boolean, values compared
Comparison === !==	matching types	Result is boolean, values & types compared

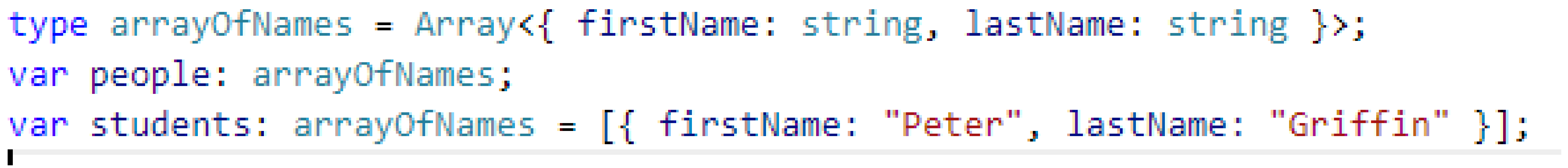
TYPE ALIASES

Type Aliases are resuable custom data types in TypeScript

```
type arrayOfNames = Array<{ firstName: string, lastName: string }>;
var people: arrayOfNames;
```

WALKTHRU

Type Aliases



```
type arrayOfNames = Array<{ firstName: string, lastName: string }>;
var people: arrayOfNames;
var students: arrayOfNames = [{ firstName: "Peter", lastName: "Griffin" }];
```

INFERRED TYPING

TypeScript can infer the data type based on its assigned value
This type will be applied as if it was an actual Type Annotation

```
var inferredString = "string";
var inferredNumber = 5;
```

TYPE SHAPES AND COMPATIBILITY

Type compatibility in TypeScript is based on structural subtyping

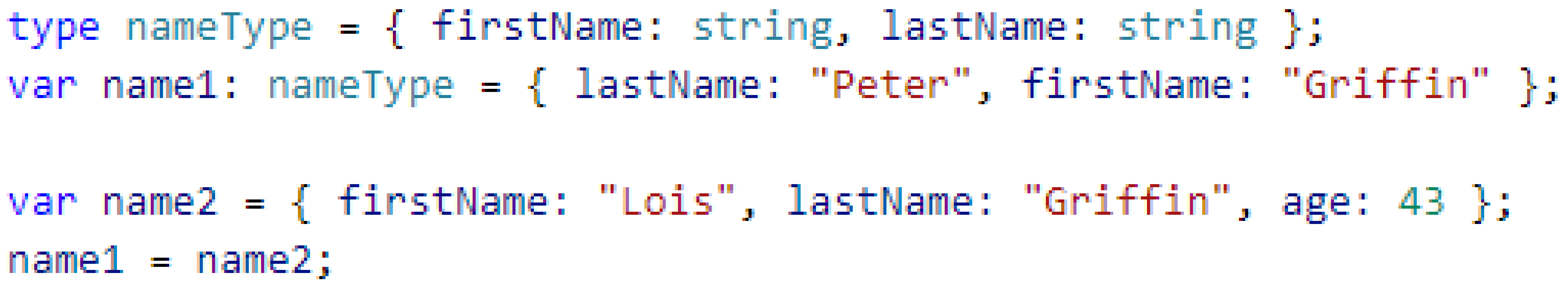
Structural typing is a way of relating types based solely on their members

```
interface Named { name: string; }

class Person { name: string; }

// OK, because of structural typing
var p: Named = new Person();
```

In nominally-typed languages like C# or Java, the equivalent code would be an error because the Person class does not implement the Named interface

WALKTHRU

Type Inference, Shapes, and Duplicate Identifiers



```
type nameType = { firstName: string, lastName: string };
var name1: nameType = { lastName: "Peter", firstName: "Griffin" };

var name2 = { firstName: "Lois", lastName: "Griffin", age: 43 };
name1 = name2;
```

ANY QUESTIONS?