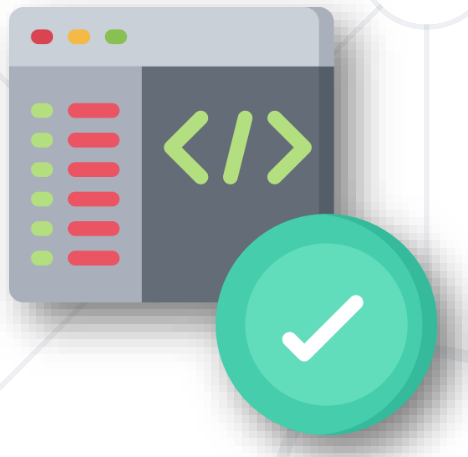
Coding Standards. Code Analysis

Optimizing Software Quality



SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#QA-Auto-BackEnd

Table of Contents

Coding Standards

Introduction

What are Coding Standards?

- Set of **guidelines** and **best practices** for writing code
- Designed to ensure **consistency**, **readability**, and **maintainability** across a codebase
- The main **goal** is to make the **code accessible** and **understandable** to **all team members**, regardless of when they join the project
- Lead to a **reduction** in **technical debt** and makes software **easier** to **manage** and **update** over time

Examples

Following Standards

```
class Program
{
    static void Main(string[] args)
    {
        int[] numbers = {1, 2, 3, 4, 5};
        CalculateAndPrintSumOfFirstAndLastElements(numbers);
    }

static void CalculateAndPrintSumOfFirstAndLastElements(int[] numbers)
    {
        int firstElement = numbers[0];
        int lastElement = numbers[numbers.Length - 1];
        int sum = firstElement + lastElement;
        Console.WriteLine($"Sum of first and last elements: {sum}");
    }
}
```

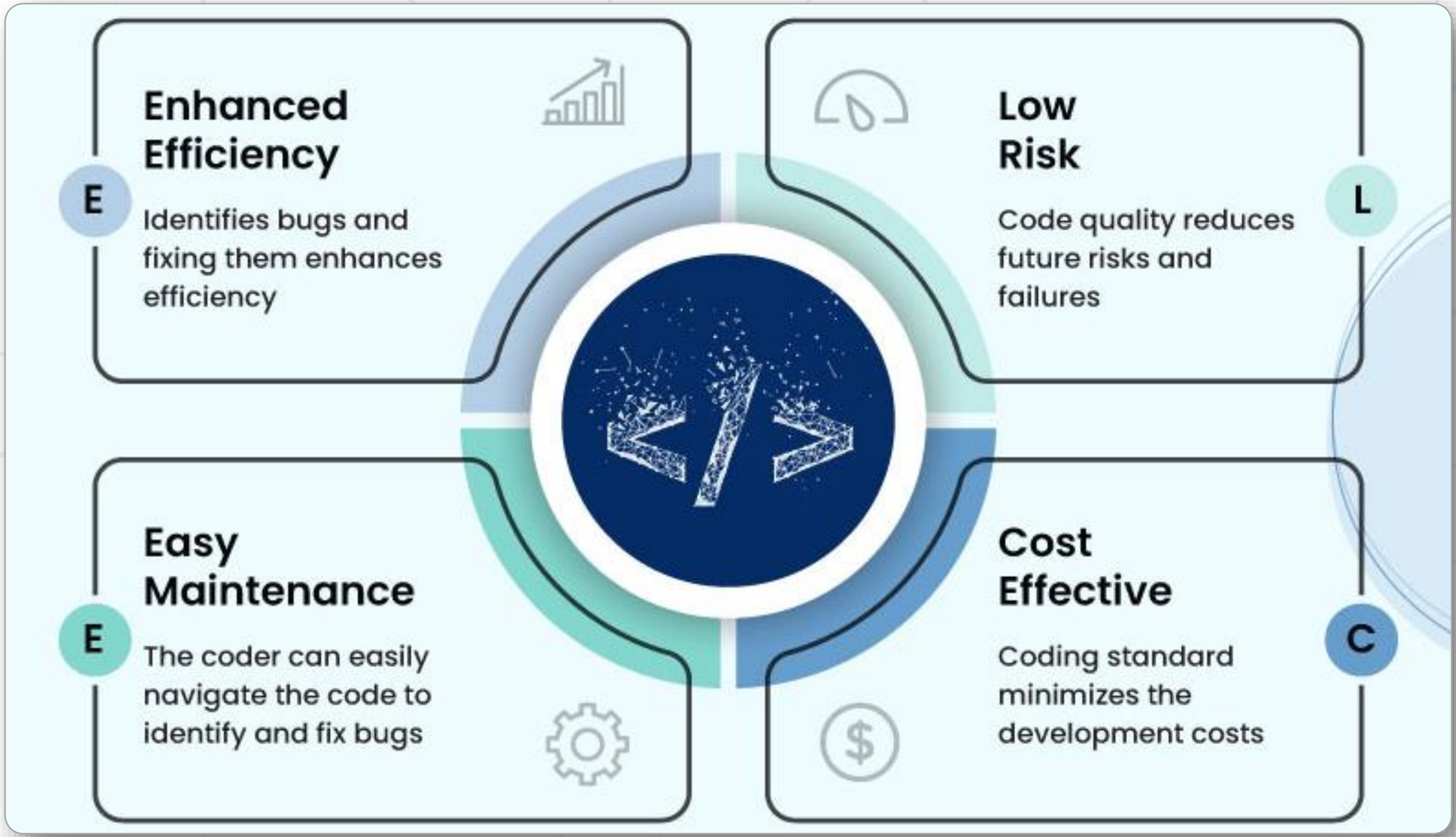
```
class program{
    static void Main(string[] args){
    int[] arr={1,2,3,4,5};calc(arr);}
    static void calc(int[] arr){
    int first=arr[0],last=arr[arr.Length-1];
    Console.WriteLine(first+last);}
}
```

Not Following Standards

Importance of Coding Standards

- The significance of coding standards is underscored by compelling **real-world statistics**:
  - Websites that **load in five seconds** retain visitors **70% longer** than those taking nineteen seconds to load
  - A mere **100-millisecond decrease** in website load time can lead to a nearly **7% reduction** in conversion rates
  - Approximately **79% of online shoppers** state they are **less likely to revisit a website** that suffers from **poor performance**

Benefits of Coding Standards



E

Enhanced Efficiency

Identifies bugs and fixing them enhances efficiency

L

Low Risk

Code quality reduces future risks and failures

E

Easy Maintenance

The coder can easily navigate the code to identify and fix bugs

C

Cost Effective

Coding standard minimizes the development costs

Common Coding Standards

- PEP 8:
  - Python Enhancement Proposal
  - The **official guide** for Python code formatting
  - It helps maintain **readability** and **consistency** in codebases
  - Covers **naming styles**, **code layout**, and **best practices**, essential for clear and collaborative Python programming
  - **Widespread adoption** ensures Python code is universally understood and easy to manage

Common Coding Standards

- Oracle's Java Code Conventions:
  - Well-established **guidelines** for writing **Java code**
  - Aims to improve the **readability** and **uniformity** of Java code among developers
  - Addresses **naming conventions**, **file organization**, and best **practices** in code structure
  - Promotes **universally comprehensible code**, easing collaboration and maintenance

Common Coding Standards

- .NET Coding Conventions:
  - Guidelines **provided by Microsoft** for writing C# code as part of the **.NET framework documentation**
  - Enhances the **clarity** and **consistency** of C# code across diverse development environments
  - Discusses best practices for **naming**, **structuring**, and **documenting** C# code
  - Facilitates **maintainable** and **understandable** code, improving team collaboration and **code longevity**

Common Coding Standards

- Several style guides for JavaScript, but one notable example is the Airbnb JavaScript Style Guide, which is **widely accepted:**
  - Aims to maintain **readable** and **coherent code** in a language that **doesn't enforce much structure**
  - Offers **guidelines** on **conventions**, **syntax**, and **best practices** tailored to **JavaScript's dynamic nature**
  - Promotes **code quality** and **consistency**, essential for scalable and maintainable JavaScript codebases

Coding Standards Best Practices

- Essentially, coding standards best practices suggest that code **quality is good** if:
  - The **code does exactly** what it is **supposed to do**
  - Maintains **consistent style**
  - **Easily understandable**
  - **Well documented**
  - **Tested**
- According to a study on Software Defect Origins and Removal Methods, employing **good coding practices** along with thorough **debugging** and **testing** can result in identifying **35% more bugs**

Do's and Don'ts

- **Code Comments**
  - It helps in understanding the code
- **Use Indentation**
  - Follow a clear and consistent style of indentation
- **Grouping Code**
  - Group the tasks in several code blocks or functions

- **Avoid Comments**
  - Ensure not to comment unnecessarily
- **Don't Repeat**
  - While coding, write codes to avoid repetition
- **Avoid Nesting**
  - Avoid deep nesting to decipher the code easily

Code Analysis

Dynamic vs. Static. Implementing Coding Standards

What is Code Analysis?

- **Detailed review** of a program's code to find errors, security breaches, and other issues
- Ensures that the code **meets quality, security, product specifications** at all stages, including **coding, testing, maintenance**, etc.
- The main goal is to **identify and fix possible errors**, before they cause real-world problems
- Two main **types**:
  - **Static** - Reviews code without execution to find errors
  - **Dynamic** - Tests code during execution to identify runtime issues

Dynamic Code Analysis

- Identifies potential **security threats, performance issues**
- Involves **feeding data** into the software and observing its behavior to uncover any issues
- Essential for **testing and debugging** throughout the software development process
- Employs **debuggers, profilers, and runtime analysis tools** to facilitate the analysis
- Selenium, Appium

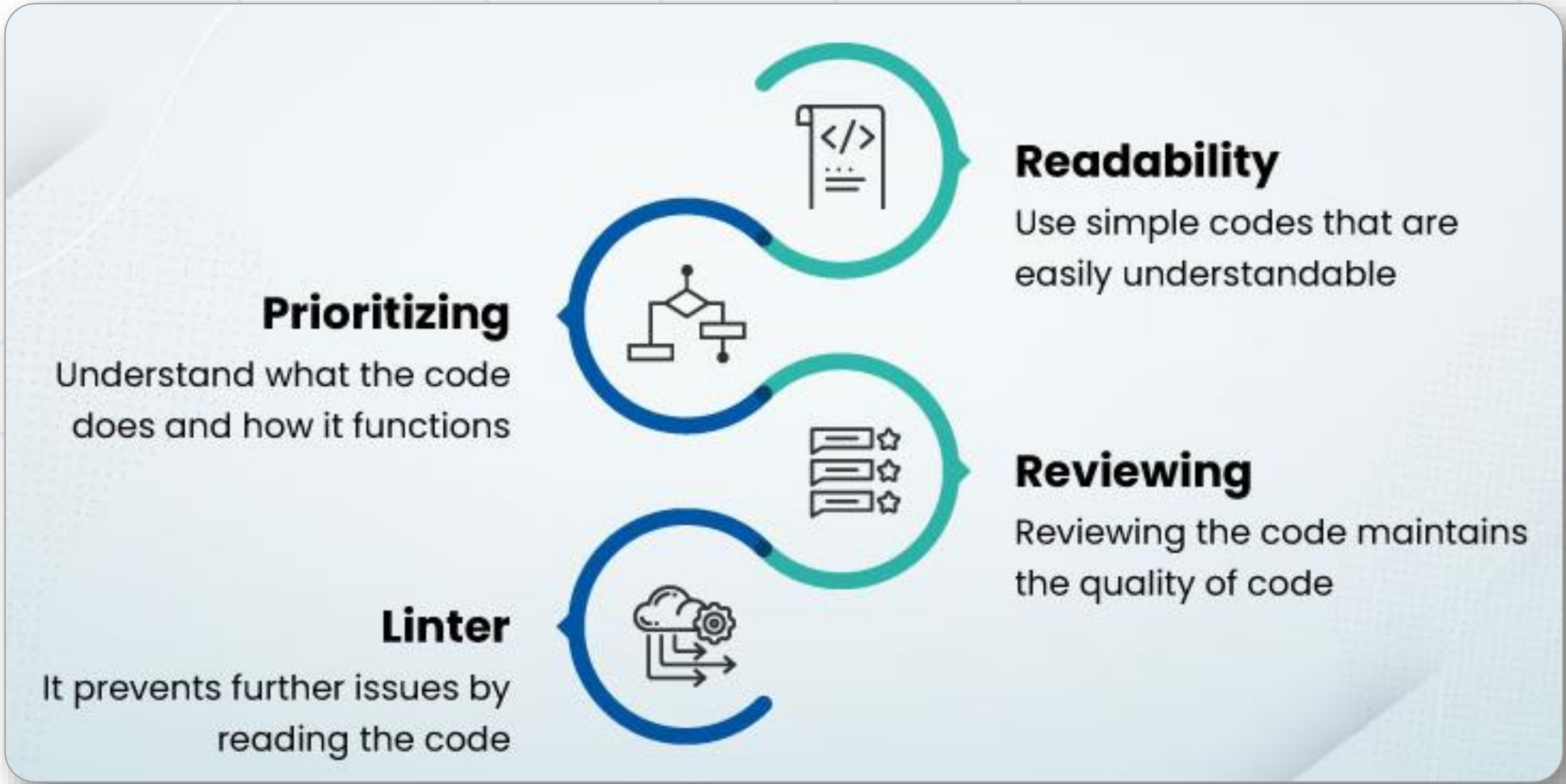
Static Code Analysis

- Detects **coding mistakes**
- Conducted during the **Code Review stage** of the Software Development Lifecycle (SDLC)
- Employs tools to examine **'static' (non-running) source code**
- **Helps security analysts** pinpoint areas of code that require closer inspection
- Aims to identify and **resolve vulnerabilities early**, ensuring **code meets standards** and **reducing future debugging needs**

Coding Standards and Static Code Analysis

- Coding standards **set the stage for quality**
- Static code analysis **enforces and validates** these standards
- Standards **prevent bad practices**
- Analysis tools **catch discrepancies**
- **Together**, they ensure the code is not just **correct**, but also **clean** and **easy to work** with
- This teamwork leads to **code that's high quality**, **consistent**, **easy to maintain**, and **less prone to bugs**

How to Implement Coding Standards



Readability
Use simple codes that are easily understandable

Prioritizing
Understand what the code does and how it functions

Reviewing
Reviewing the code maintains the quality of code

Linter
It prevents further issues by reading the code

Linters

- The term **lint**, or a **linter**, is a **static code analysis tool**
- **Used** to:
  - Identify where your **code deviates** from defined **coding standards**
  - Point out instances where **outdated** or **potentially problematic language** features are used
  - Highlight **general** programming **malpractices**
  - Detect **inconsistencies in the structure** of the code

ESLint

JS Oddities

- **JS** is a language **well known** to have a lot of "**quirks**"
- This often trip up people new to the language, even experienced developers coming from other languages
- For **example**, using the == operator, instead of === allows **types to be coerced** into their **truthy** and **falsy** equivalents
- **Oftentimes** this is **not** what is **intended** when comparing a string and a number value, and can be a **common source of errors**

```
0 == "0" //true
0 == [ ] //true
//so, if
0 == "0"
//and
0 == [ ] true
//then
"0" == [ ]
//nope, it's false
```

How a Linter Would Help?

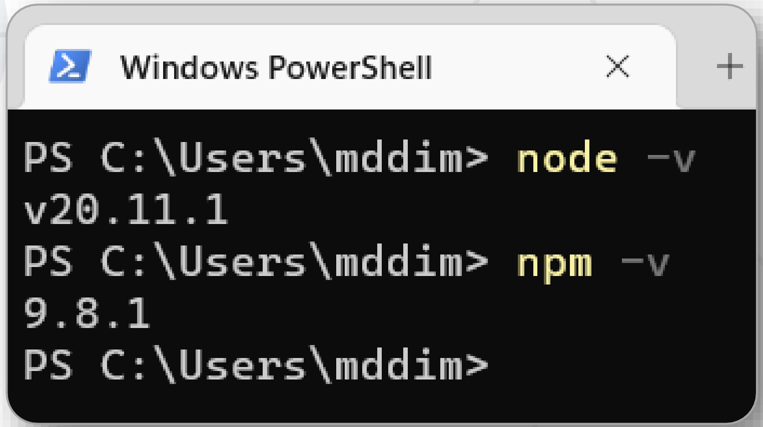
- A linter will allow to **apply a rule** that either **warns** or **prohibits** the usage of the **== operator** and guide every team member to be explicit with their comparisons
- This introduces **better consistency** across the codebase
- It allows **any developer / QA** to navigate different parts of the codebase and **quickly read and understand** what the code is designed to do
- This is just **one example** of any number of **potentially unlimited rules** that can be enabled for a linter
- You can even **write rules yourself** if you want

What is ESLint?

- Static code analysis tool used primarily to identify problematic patterns
- It aims to make code more consistent and prevent bugs through:
  - **Automated Code Review**: Scans JS code for common errors and potential issues
  - **Customizable Rules**: Can be configured with different rules to enforce a particular coding style
  - **Plugin System**: Supports a wide range of plugins for additional rules or frameworks
  - **Fixing Capabilities**: Can automatically fix many of the issues it detects, making code corrections easier
  - **Integration**: Can be integrated into development workflows, including editors, build systems, and continuous integration pipelines

Prerequisites

- You will need to have **Node.js installed** on your machine and available from your terminal
- Installing Node will automatically install **npm** as well
- Open up your terminal of choice
- If you see version numbers when running the **two commands** from the picture (your numbers will likely be different than this example) then you are **ready to go**



```
PS C:\Users\mddim> node -v
v20.11.1
PS C:\Users\mddim> npm -v
9.8.1
PS C:\Users\mddim>
```

Initializing the Project

- Let's start by **initializing a new npm project**
- Create a **new folder** for your project where all the files will be stored
- **Navigate into it** using your **command-line interface**
- Run the following command:

```
npm init -y
```

- This command **initializes a new npm project** with **default settings**, creating a **package.json** file in your current directory
- The **-y flag** automatically **answers 'yes'** to all the prompts, saving you from having to manually **set up the configuration**

Initializing the Project

- **Open** your project in VSC
- As you can see the **package.json** file is **present**
- **Create a file** in your root directory called **script.js** with the following code:

```
const person = {
  name: 'Jen',
  name: 'Steve',
}

let some_text = undefined;

function printHelloWorld() {
  console.log("hello world!");
};

printHelloWorld()
```

Initializing the Project

- The code from previous slide is **perfectly valid JS**
- You can **verify by** running:

```
node script.js
```

- And you will get the **output "hello world!"**
- However, **despite being valid** there are a **number of problems** that might prevent code like this from passing a review

Mistakes

- **Person**: is assigned two names, one overwrites the other
- **Semicolons are inconsistent:** Some lines have them, others don't
- **Quotations are inconsistent:** Some code uses single, others double
- **some_text:** is written in snake_case instead of camelCase
- **person** and **some_text:** are never actually used. Why declared?

- This code could be sent back to the developer, with this written list saying "**please fix**", but something as basic as this can **easily** be **identified** with a static analysis tool like **ESLint**

Installing ESLint

- Add ESLint to the project with the following command:

```
npm install eslint --save-dev
```

- At this point you have the option of running the command:

```
npx eslint --init
```

- It will take you through some questions in your terminal about what kind of project you are making and what tools you are using
- This is a great way to get started, but our goal is to **understand each piece of the configuration** as we implement it, **we are going to create our configuration file from scratch**

Configure ESLint

- Create a `.eslintrc.json` file in your root directory (notice that our config filename begins with a . to indicate it is a **hidden file**):

```json
{
  "env": {
    "browser": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 2021
  },
  "rules": {
    "quotes": ["error", "double"],
    "semi": ["error", "always"]
  }
}
```

Config fields: env

- What **each one of the fields** in the config **does**:
  - **env**

```
"env": {
    "browser": true
  }
```

    - **Specifies the environment** we are planning to run our code in
    - When we say **browser**, ESLint will not throw an error if we try to use a DOM method like document.querySelector()
    - Another common **env** value is **node**

Config fields: extends

- **extends**

```
"extends":
"eslint:recommended"
```

  - This option allows us to **inherit** from **existing lists of rules**
  - ESLint provides a list of default recommended rules
  - If there are any you disagree with, they can be disabled manually in the rules field on the config

Config fields: parserOptions

- **parserOptions**

```
"parserOptions": {
    "ecmaVersion": 2021
  }
```

  - The **ecmaVersion** property tells ESLint which ECMA version of Javascript you are targeting
  - For example if you use a value for **2015** it will **throw an error** if you try to use syntax like **const** or **let** instead of var
  - Setting it to **2016** would allow you to **use them**

Config fields: rules

- Rules

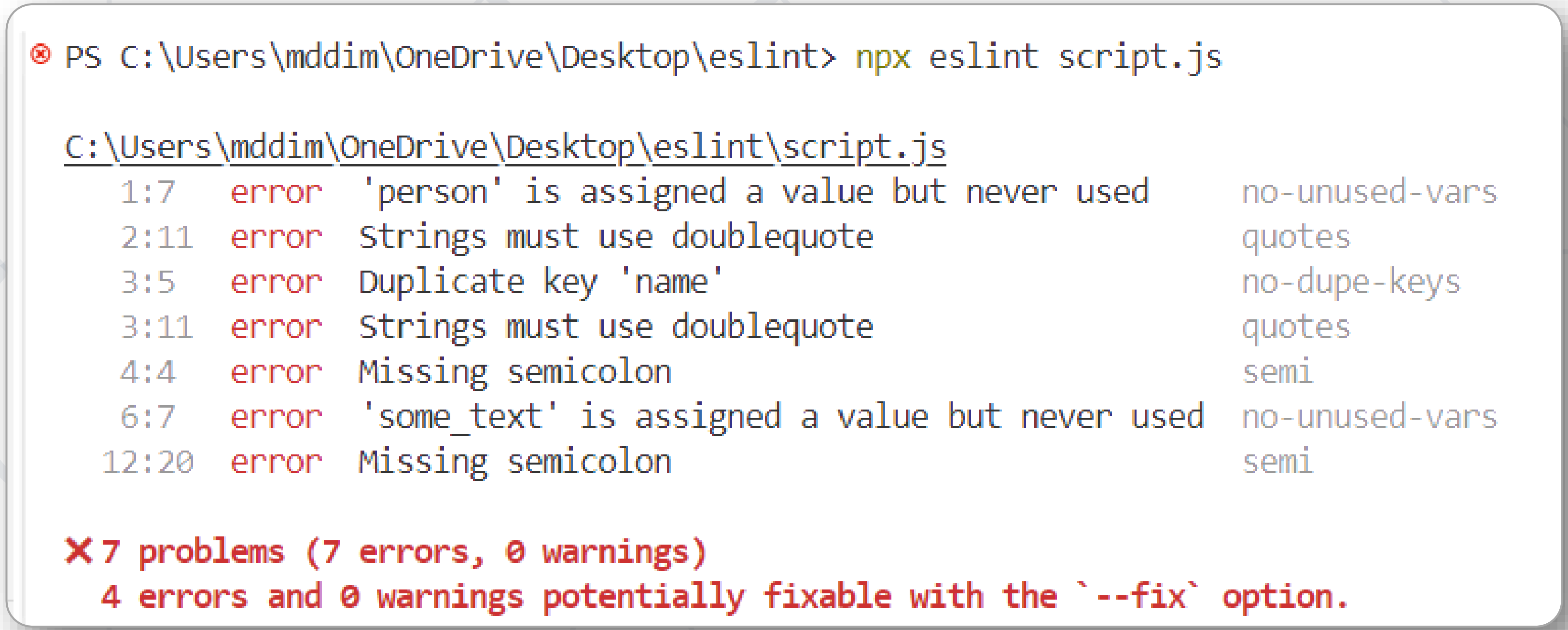
```
"rules": {
     "quotes": ["error", "double"],
     "semi": ["error", "always"]
  }
```

  - This is where we **manually configure any rules** we would like to apply in our project, and whether we want to show a warning or throw an error
  - Tools can be set to listen for ESLint errors and cancel if they are encountered
  - We use the default **eslint:recommended** set of rules, but also **enforce** that **semicolons must always be used at the end of lines**, and all developers on the team **use double quotes instead of single**

Linting the Project

- With this **configuration in place**, let's **run ESLint on our script.js** file with the following command:

```
npx eslint script.js
```

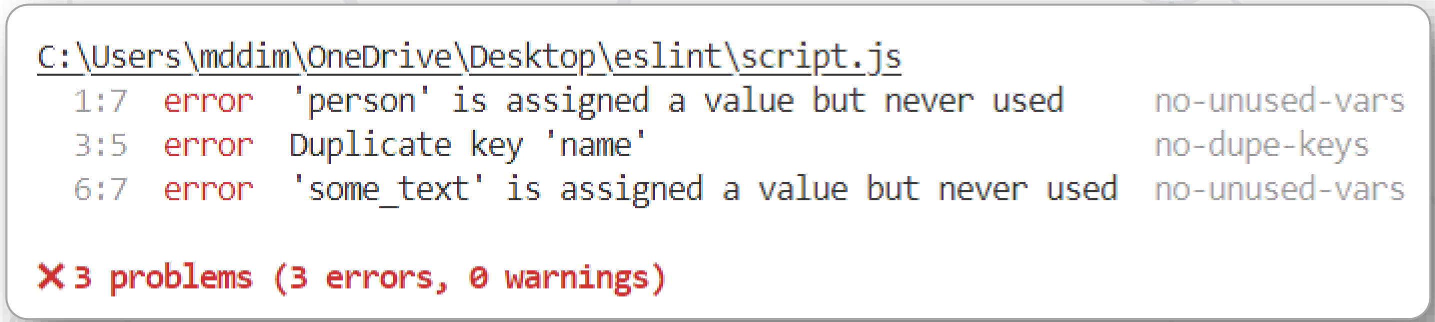


```
⊗ PS C:\Users\mddim\OneDrive\Desktop\eslint> npx eslint script.js

  C:\Users\mddim\OneDrive\Desktop\eslint\script.js
     1:7   error  'person' is assigned a value but never used     no-unused-vars
     2:11  error  Strings must use doublequote                    quotes
     3:5   error  Duplicate key 'name'                            no-dupe-keys
     3:11  error  Strings must use doublequote                    quotes
     4:4   error  Missing semicolon                               semi
     6:7   error  'some_text' is assigned a value but never used  no-unused-vars
    12:20  error  Missing semicolon                               semi

  ✕ 7 problems (7 errors, 0 warnings)
    4 errors and 0 warnings potentially fixable with the `--fix` option.
```

Linting the Project

- ESLint has provided us with the **information needed** to **correct the errors** in our code
- Not only does it **inform** us of the **issues**, it even knows how to **fix some** of the **more basic syntax issues** like quotes and semicolons
- Run the command:

```
npx eslint script.js --fix
```



```
C:\Users\mddim\OneDrive\Desktop\eslint\script.js
  1:7  error  'person' is assigned a value but never used     no-unused-vars
  3:5  error  Duplicate key 'name'                            no-dupe-keys
  6:7  error  'some_text' is assigned a value but never used  no-unused-vars

✖ 3 problems (3 errors, 0 warnings)
```

Linting the Project

- The problems with obvious solutions have been fixed
- Check out script.js and see for yourself the file has been edited
- The other values don't have obvious solutions
- Deciding whether or not to use person is more of a program logic decision than a syntax error
- Similar, ESLint can't be sure which of the two names is correct

Linting the Project

- So we decide to refactor our script.js code so it looks like this:

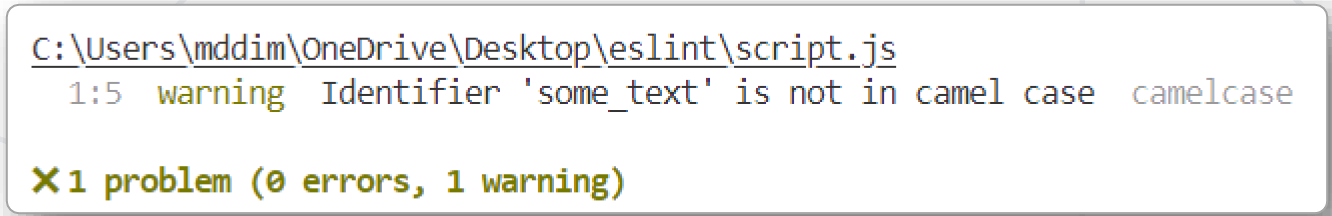
```
let some_text = "hello world!";

function printHelloWorld() {
  console.log(some_text);
};

printHelloWorld();
```

- When we run **`npx eslint script.js`** again we see no output
- No output is good! It means there are no errors

Linting the Project

- Except **`some_text`** is still using **snakeCase** instead of **camelCase**
- Casing in variable names is a rule that exists called camelcase, it's just not enabled by default
- Let's turn it on in our config file **`.eslintrc.json`**:
- We decide that enforcing camelCase isn't as important as making sure to use all the variables we declare, so we set it to warn instead of error

```
"rules": {
    "quotes": ["error", "double"],
    "semi": ["error", "always"],
    "camelcase": "warn"
  }
```



```
C:\Users\mddim\OneDrive\Desktop\eslint\script.js
  1:5  warning  Identifier 'some_text' is not in camel case  camelcase

✕ 1 problem (0 errors, 1 warning)
```

Extending Configurations (Airbnb)

- You can easily inherit from third party ESLint configurations that you've installed into your project
- One of the most famous examples is **eslint-config-airbnb** based on the set of linting rules used by Airbnb software developers
- To apply the same sets of rules they use, first install the plugin:

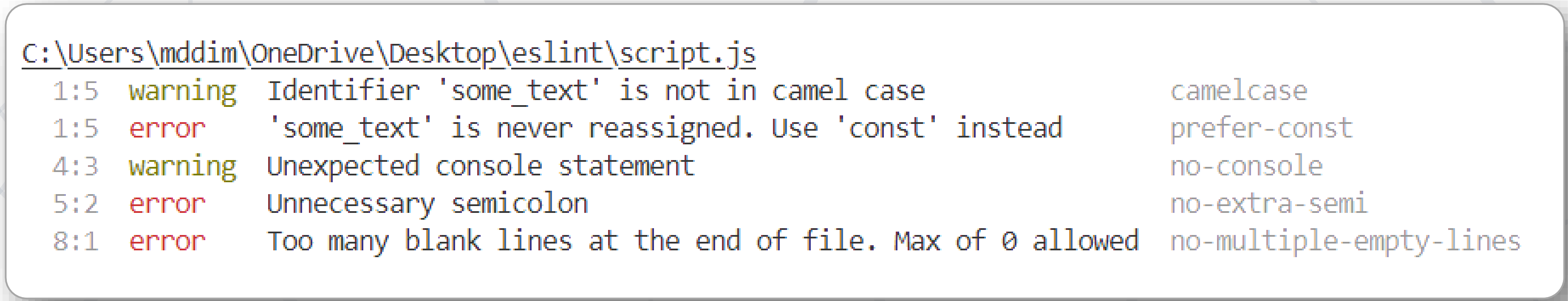
```
npm install eslint-config-airbnb --save-dev
```

- Now add the plugin to the list of configurations we are extending in our config file:

```
"extends": ["eslint:recommended", "airbnb"]
```

Airbnb Standards

- Now when we run npx ESLint script.js we will discover that our program that previous met our own standards, no longer meets the higher standards of Airbnb

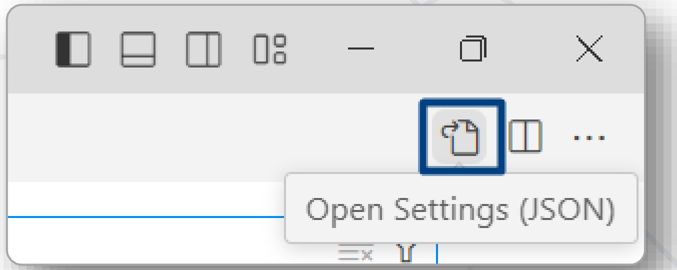


```
C:\Users\mddim\OneDrive\Desktop\eslint\script.js
  1:5  warning  Identifier 'some_text' is not in camel case              camelcase
  1:5  error    'some_text' is never reassigned. Use 'const' instead     prefer-const
  4:3  warning  Unexpected console statement                             no-console
  5:2  error    Unnecessary semicolon                                    no-extra-semi
  8:1  error    Too many blank lines at the end of file. Max of 0 allowed  no-multiple-empty-lines
```

Editor Integration (VS Code)

- ESLint can be integrated into your workflow to enable you to **see errors as you type them**
- Install the ESLint extension for VS Code and enable it
- Next we need to open VS Code's settings.json file
- You can find it in the File > Preferences > Settings menu
- There is a link in the upper right corner to access **settings.json** directly
- Add the following:

```
"eslint.validate": ["javascript"]
```



Static Code Analyzers for .NET

Visual Studio

- In the **C# development ecosystem**, especially with IDE like **Visual Studio**, many features that are typically provided by linters in other languages are **already built-in** their analysis tools:
  - **IDE Features** - comprehensive code analysis tools
  - **Automatic Formatting** - Enforce code styling and formatting automatically
  - **Roslyn Analyzers** - Part of .NET, they extend built-in IDE analysis capabilities during compilation
  - **EditorConfig** - Allows for consistent coding styles across various environments
  - **Less Need for Linters** - The combination of these features reduces the necessity for separate linting tools in C#

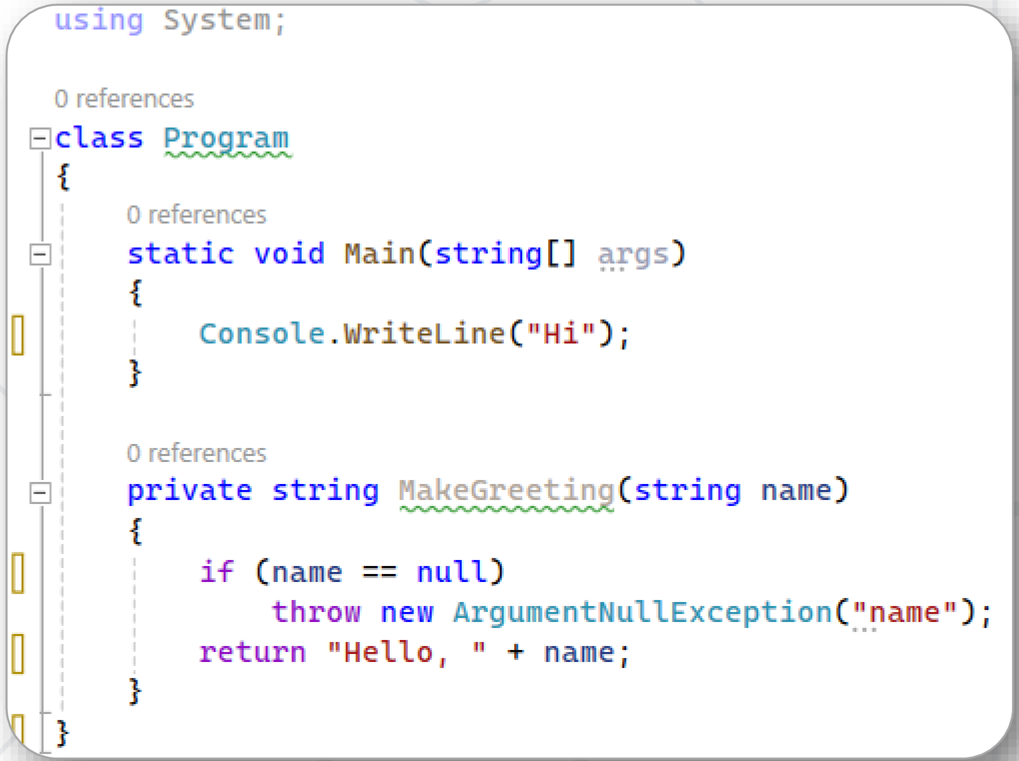
Analyzers

- .NET Compiler Platform (Roslyn) Analyzers inspect C# or Visual Basic code for style, quality, maintainability, design, and other issues
- This inspection or analysis happens during design time in all open files
- **Analyzers** are **divided** into the **three groups**:
  - **Code style analyzers** (built into Visual Studio)
    - **Code Style Rules** are divided into:
      - **Language rules**
      - **Formatting rules**
      - **Naming rules**

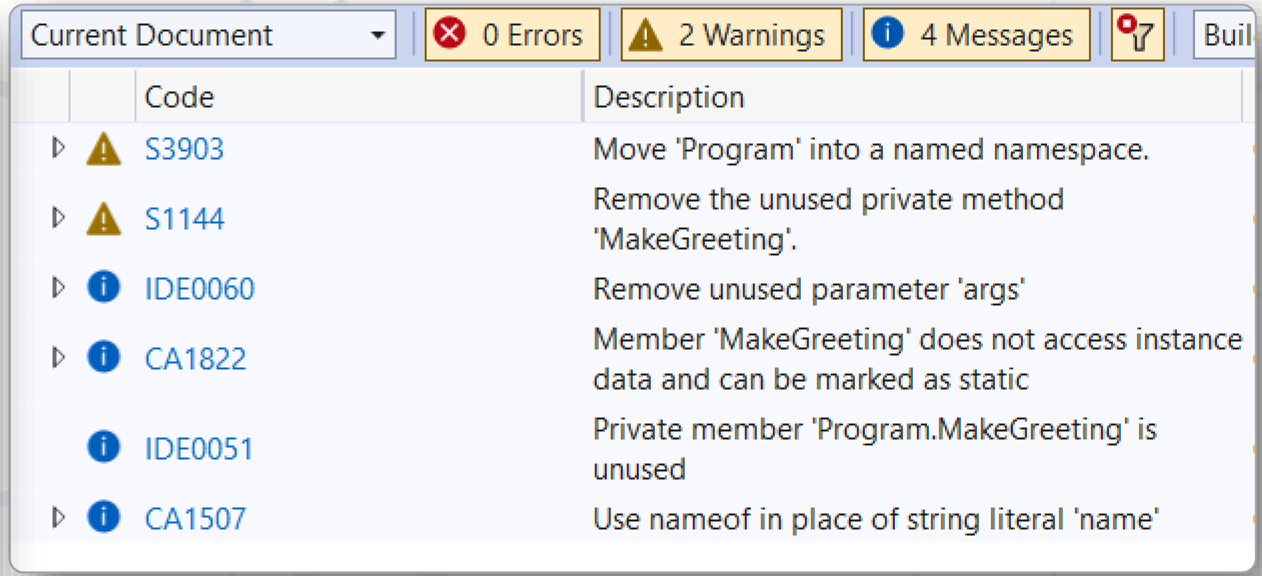
Analyzers

- **Code quality analyzers** (now included with the .NET 5 SDK and enabled by default)
  - [Code Quality Rules]
- **External analyzers** such as StyleCop, Roslynator, XUnit Analyzers, and Sonar Analyzer as a NuGet package or a Visual Studio extension

Built-In Analyzers



```
using System;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hi");
    }

    0 references
    private string MakeGreeting(string name)
    {
        if (name == null)
            throw new ArgumentNullException("name");
        return "Hello, " + name;
    }
}
```

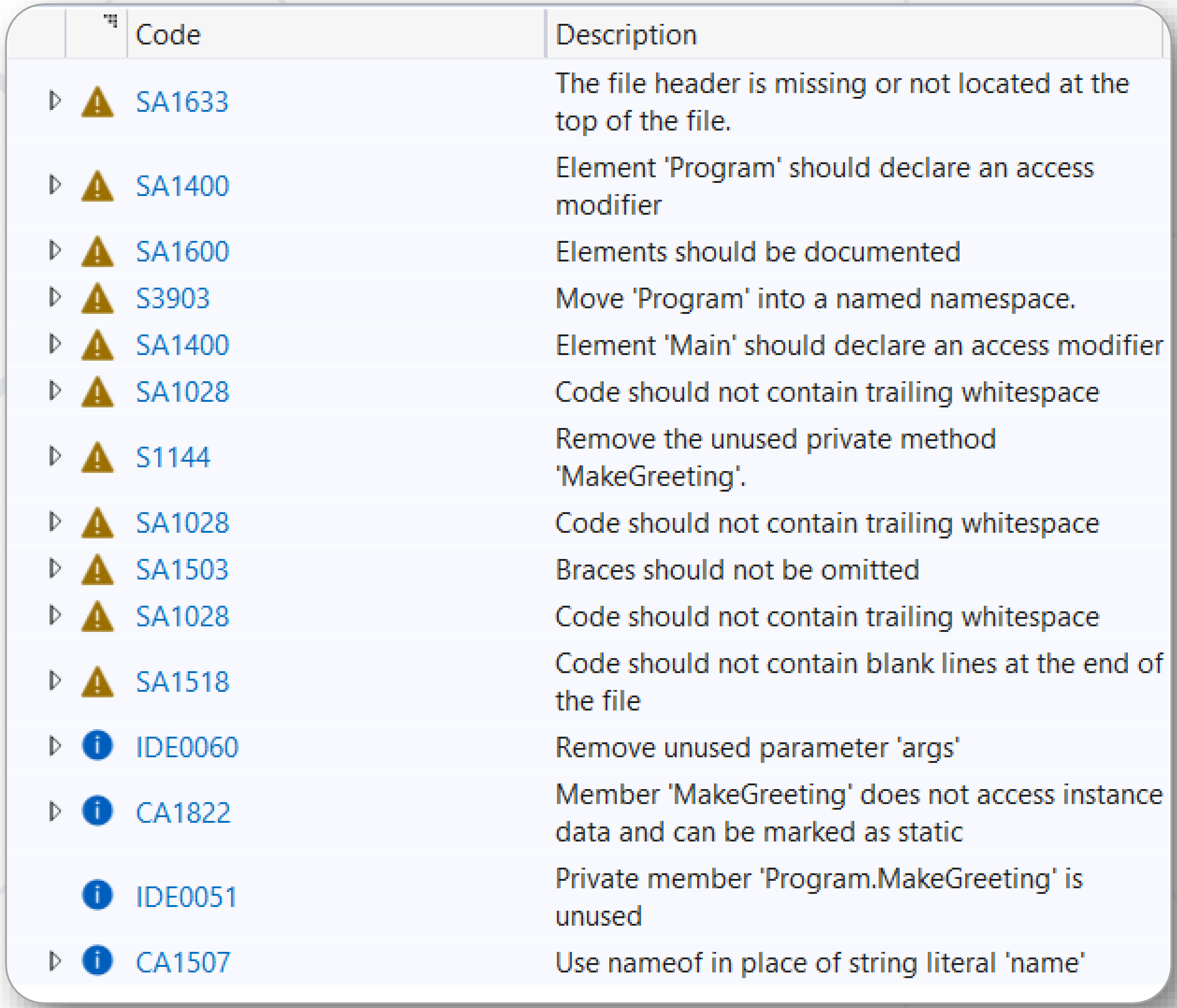


Code	Description
S3903	Move 'Program' into a named namespace.
S1144	Remove the unused private method 'MakeGreeting'.
IDE0060	Remove unused parameter 'args'
CA1822	Member 'MakeGreeting' does not access instance data and can be marked as static
IDE0051	Private member 'Program.MakeGreeting' is unused
CA1507	Use nameof in place of string literal 'name'

- Visual Studio comes with built-in analyzers that can automatically detect a range of code issues, from syntax errors to certain code quality and security issues

StyleCop

- Usually the built-in analyzers are enough
- But let's try StyleCop with the same code
- Install StyleCop.Analyzers via Nuget Packages
- The errors found by StyleCop are marked SA….
- So, additional tools can be added for broader coverage

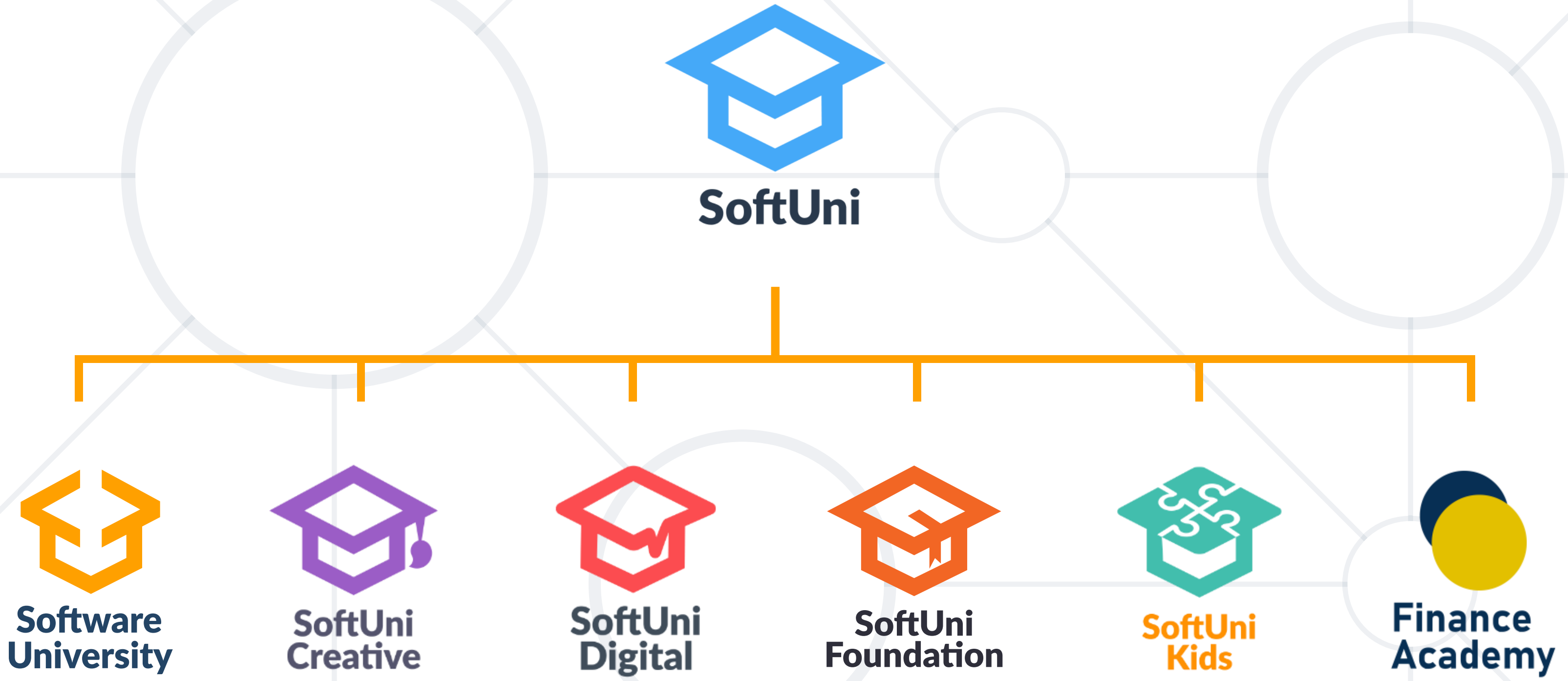


Code	Description
⚠ SA1633	The file header is missing or not located at the top of the file.
⚠ SA1400	Element 'Program' should declare an access modifier
⚠ SA1600	Elements should be documented
⚠ S3903	Move 'Program' into a named namespace.
⚠ SA1400	Element 'Main' should declare an access modifier
⚠ SA1028	Code should not contain trailing whitespace
⚠ S1144	Remove the unused private method 'MakeGreeting'.
⚠ SA1028	Code should not contain trailing whitespace
⚠ SA1503	Braces should not be omitted
⚠ SA1028	Code should not contain trailing whitespace
⚠ SA1518	Code should not contain blank lines at the end of the file
ⓘ IDE0060	Remove unused parameter 'args'
ⓘ CA1822	Member 'MakeGreeting' does not access instance data and can be marked as static
ⓘ IDE0051	Private member 'Program.MakeGreeting' is unused
ⓘ CA1507	Use nameof in place of string literal 'name'

Summary

- Coding Standards: Guidelines for writing code, enhances readability, maintainability
- Common Coding Standards: Widely accepted practices
- Code Analysis: Reviewing code quality
- Static vs. Dynamic:
- ESLint: JavaScript code linter
- Tools for C# quality checks

Questions?


SoftUni
Software University
SoftUni Creative
SoftUni Digital
SoftUni Foundation
SoftUni Kids
Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank

Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

License

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg