# 2. 代码质量

## 课后作业讲解

给定一份澳洲个人所得税表，根据输入的个人收入，计算出对应的个人所得税。

| Taxable income | Tax on this income |
| --- | --- |
| 0 – $18,200 | Nil |
| $18,201 – $45,000 | 19 cents for each $1 over $18,200 |
| $45,001 – $120,000 | $5,092 plus 32.5 cents for each $1 over $45,000 |
| $120,001 – $180,000 | $29,467 plus 37 cents for each $1 over $120,000 |
| 180,000 and over | $51,667 plus 45c for each $1 over $180,000# |

### 1a. 使用 `if`

```
// Close: 税表, Close: 计算方式 function calculateTaxIf(income) { if (incom
e > 180000) { return 51667 + 0.45 * (income - 180000); } if (income > 120
000) { // 查找 return 29467 + 0.37 * (income - 120000); // 计算 } if (inco
me > 45000) { return 5092 + 0.325 * (income - 45000); } if (income > 1820
0) { return 0.19 * (income - 18200); } return 0; }
```

### 1b. 使用 `if`

```
function calcualteTaxIf(income) { if (income > 0 && income <= 18200) {
return 0; } if (income > 18200 && income <= 45000) { return (income -
18200) * 0.19; } if (income > 45000 && income <= 120000) { return 5092 +
(income - 45000) * 0.325; } if (income > 120000 && income <= 180000) {
return 29467 + (income - 120000) * 0.37; } if (income > 180000) { return
51667 + (income - 180000) * 0.45; } return }
```

### 1c. 使用 `if-else`：

```
function calculateTaxIfElse(income) { if (income <= 18200) { return 0; } else if (income <= 45000) { return (income - 18200) * 0.19; } else if (income <= 120000) { return 5092 + (income - 45000) * 0.325; } else if (income <= 180000) { return 29467 + (income - 120000) * 0.37; } else { return 51667 + (income - 180000) * 0.45; } return 0; }
```

## 2. 使用 `switch`:

```
function calculateTaxSwitch(income) { switch (true) { case income <= 18200: return 0; case income <= 45000: return (income - 18200) * 0.19; case income <= 120000: return 5092 + (income - 45000) * 0.325; case income <= 180000: return 29467 + (income - 120000) * 0.37; default: return 51667 + (income - 180000) * 0.45; } }
```

💡 高级的写法使用并不代表好的代码!

## 3. 使用字典数据结构:

```
// 根据税表和收入，计算个人说的税 const TAX_TABLE = [ { min: 0, max: 18200, base: 0, rate: 0 }, { min: 18200, max: 45000, base: 0, rate: 0.19 }, { min: 45000, max: 120000, base: 5092, rate: 0.325 }, { min: 120000, max: 180000, base: 29467, rate: 0.37 }, { min: 180000, max: Infinity, base: 51667, rate: 0.45 } ]; function calculateTaxDictionary(taxTable, income) { const row = taxTable.find(({ min, max }) => income > min && income <= max); const tax = row.base + (income - row.min) * row.rate; return tax; }
```

MYOB | XERO
政府突然提出新财年税务改动

| Taxable Income | Tax On This Income |
| --- | --- |
| 0 to $18,200 | Nil |
| $18,201 to $45,000 | 19c for each $1 over $18,200 |
| $45,001 to $200,000 | $5,092 plus 30% for each $1 over $45,000 |
| $200,001 and over | $51,592 plus 45c for each $1 over $200,000 |

## 4. 定义 Level 对应 Tax 的计算方式，通过 Income 找 Level

```
// Open: 计算的责任 const formula = { // 计算， 在这里 key 是查找的结果 level1: (income) => 0, level2: (income) => (income - 18200) * 0.19, level3: (income) => (income - 45000) * 0.30 + 5092, level4: (income) => (income - 200000) * 0.45 + 51592, // 虽然现在所有的计算都是基于 base 和 rate // 未来可能会出现某个阶梯的计算方法完全不同的情况，所以把每个阶梯的计算公式单独列出来 }; // Open: 查找的责任 const getLevel = (income) => { // 查找 if (income <= 18200) { return "level1"; } else if (income <= 45000) { return "level2"; } else if (income <= 20000) { return "level3"; } else { return "level4"; } }; function calculateTax(income) { const level = getLevel(income); return formula[level](income); }
```

### 可读性

- `if-else`：直观但多个条件和嵌套可能使代码难以阅读。
- `switch`：逻辑较为集中，但与 `if-else` 相似，多个 `case` 可能导致代码变得冗长和难以追踪。
- **字典数据**: 最易于阅读，因为所有信息都集中在一个数据结构中。

### 可维护性

- `if-else`：需要手动更新多个地方的代码，容易出错。
- `switch`：与 `if-else` 类似，税率或阈值的变动需要在多个 `case` 中进行更新。
- **字典数据**: 单一数据结构使得维护变得相对简单和直接。

### 可复用性

- `if-else`：需要复制整个`if-else`块并进行适当修改以实现重用。
- `switch`：和`if-else`方案类似，不容易复用，除非整个`switch`语句被封装在一个函数中。
- **字典数据**: 可以轻易地作为参数传递，更适合在不同的场景或问题中重用。

> 💡 代码质量是评价一个软件开发人员或团队水平的重要指标。高质量的代码不仅易于阅读、维护和复用，还能减少错误和漏洞的可能性。这对于项目的长期成功和团队成员之间的有效协作有着至关重要的影响。
> 作为一名软件开发领导者的角度来看，推崇高质量代码的标准能够提升整个团队的工作效率，降低维护成本，并增强产品的稳定性和可靠性。它还能提高团队的专业声誉，这在招聘优秀人才和赢得客户信任方面也是非常重要的。

## 写出来符合人类思考方式的代码

> 💡 1. Google 出来税表 → `const TAX_TABLE`
> 2. 通过收入找到相应的税率 → `find row from TAX_TABLE`
> 3. 计算 → `calculate based on the row`
>
> 1. Google 出来税表
> 2. 如果你的收入是 0 - 18200，要交多少税
> 3. 如果你的收入...

1. 在写代码之前，充分了解问题可以帮助你编写更符合实际需求的代码。
2. 变量、函数和类的命名应该具有描述性，并准确反映其用途或行为。
3. **Don't Repeat Yourself** - 避免重复代码。重复的代码会增加维护成本并增加出错的可能性。
4. **Keep It Simple, Stupid** - 简单通常比复杂好。简单的代码更容易理解和维护。
5. **You Ain't Gonna Need It** - 不要添加当前不需要的功能。过度设计会增加复杂性，并使未来更改更为困难。
6. 即使代码已经“工作”，也应该回顾和改进，以适应新的需求和技术。

> 💡 不要 Copy Paste！不要 Copy Paste！不要 Copy Paste！

> 💡 同一个功能实现3遍！同一个功能实现3遍！同一个功能实现3遍！

# SOLID原则 （May, 2009）

1. **单一职责原则**（Single Responsibility Principle）

   一个代码块应该只负责一项任务，使其易于理解和维护。

2. **开放封闭原则**（Open/Closed Principle）

   一个代码块应该对扩展开放，对修改封闭。当需要添加新功能时，应通过扩展现有代码，而非修改现有代码。

3. ~~**里氏替换原则**（Liskov Substitution Principle）~~

   ~~子类应能替换其基类并能透明地工作。这确保了继承的合理性，并使得类之间的关系更加灵活。~~

4. ~~**接口隔离原则**（Interface Segregation Principle）~~

   ~~客户端不应依赖于它不使用的接口。换句话说，一个类不应被迫实现它不会用到的接口。~~

5. ~~**依赖倒置原则**（Dependency Inversion Principle）~~

   ~~高层模块不应依赖于低层模块，它们都应依赖于抽象。这有助于降低模块间的耦合度。~~

6. **依赖注入原则** (Dependency Injection Principle)
   一个代码块的依赖应该由外部注入，而不是耦合在实现逻辑中

## getStops

```
function getStops(flights) { // [MEL - HKG] -> Direct if (flights.length === 1) { return 'Direct' } // [MEL - SIN, SIN - HKG] -> 1 Stop if (flights.length === 2) { return '1 Stop' } if (flights.length === 12) { // Easter Egg return 'Around The World' } // [MEL - SYD, SYD - SIN, SIN - HKG] -> 2 Stops return `${flights.length - 1} Stops` }
```

💡 Stop(s)

```
const student = { id: 1, name: 'Alice', age: 23, } const id = student.id
const key = 'id' // string const value = student[key] // 1 const value =
student['id'] // 1 // 1. 当你明确知道你需要的 key 的时候，可以通过解构，或者
obj.key // 2. 当你不知道你具体需要的 key 的时候，key 可以是一个 variable，这时
候可以通过 obj[key] const getStudentValue = (key) => student[key]
getStudentValue('age') // 23 const fruits = ['apple', 'banana']; const
apple = friuts[0] const index = 0 const value = friuts[index] // apple
const fruits = { 0: 'apple', 1: 'banana', } const key = 0 const value =
fruits[key] // apple
```

```
const getStops = (flights) => { 1: "Direct", 2: "1 Stop", 12: "Around the
world", }[flights.length] ?? `${flights.length - 1} Stops` // 这不是最优雅
的代码
```

符合人类思考方式的代码是好的代码

1. 先找的Stops数量
2. 查看这个 Stops 数量是不是特殊的
3. 如果是特殊的，用特殊的 Message
4. 如果不是特殊的，返回正常的 Message， n Stop(s)
5. 正常 Message 需要做单复数处理