认识重构

目录

基本概念

- **“重构”这个词既可以用作名词也可以用作动词。**
  - 重构（名词）：对软件内部结构的一种调整， 目的是在不改变软件可观察行为的前提下， 提高其可理解性， 降低其修改成本。
  - 重构（动词）：使用一系列重构手法， 在不改变软件可观察行为的前提下，调整其结构。
- **重构的关键在于运用大量微小且保持软件行为的步骤， 一步步达成大规模的修改。 每个单独的重构要么很小， 要么由若干小步骤组合而成。**

基本概念

- **重构与性能优化有很多相似之处： 两者都需要修改代码， 并且两者都不会改变程序的整体功能。**
- **两者的差别在于其目的：**
  - 重构是为了让代码“更容易理解，更易于修改”。 这可能使程序运行得更快， 也可能使程序运行得更慢。
  - 在性能优化时， 我只关心让程序运行得更快， 最终得到的代码有可能更难理解和维护。

整体流程

plays.json...

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

invoices.json...

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }

]
```

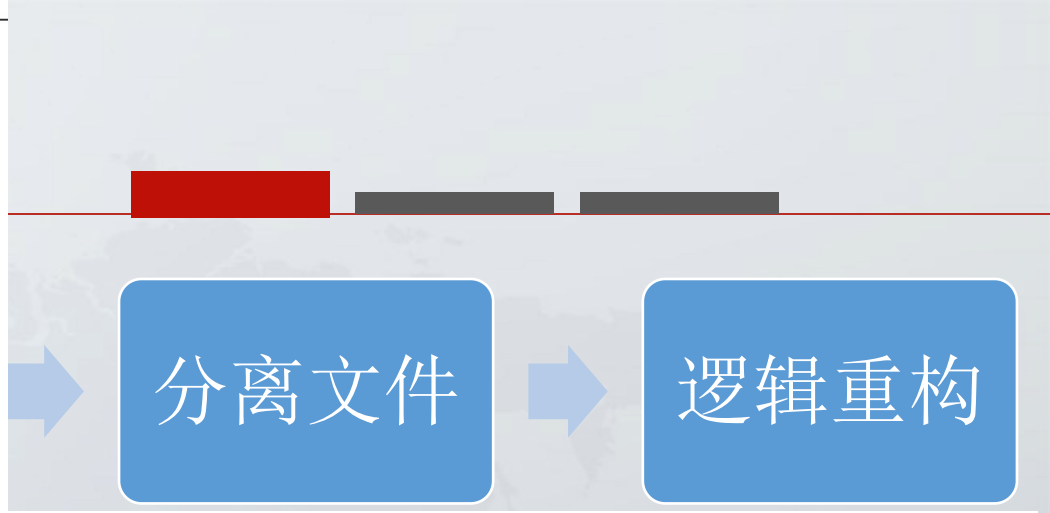
整体流程

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

整体流程

- 重构的第一个步骤永远相同： 确保即将修改的代码拥有一组可靠的测试。
- 例如：
  - statement函数的返回值是一个字符串， 就是创建几张新的账单（invoice），假设每张账单收取了几出戏剧的费用， 然后使用这几张账单作为输入调用statement函数， 生成对应的对账单（statement） 字符串。 拿生成的字符串与已经手工检查过的字符串做比对。

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audier
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${
    }

    // add volume credits
    volumeCredits += Math.max(perf.audier
    // add extra credit for every ten com
    if ("comedy" === play.type) volumeCre

    // print line for this order
    result += ` ${play.name}: ${format(th
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(tota
  result += `You earned ${volumeCredits}
  return result;
}
```



function statement...

```
function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
  case "tragedy":
    thisAmount = 40000;
    if (perf.audience > 30) {
      thisAmount += 1000 * (perf.audience - 30);
    }
    break;
  case "comedy":
    thisAmount = 30000;
    if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
    }
    thisAmount += 300 * perf.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return thisAmount;
}
```

整体流程

- 拆分阶段的目标是将逻辑分成两部分：
  - 一部分计算详单所需的数据
  - 另一部分将数据渲染成文本或HTML。

整体流程

初始程序 → 分解函数 → 拆分计算 → 分离文件 → 逻辑重构

```
function statement (invoice, plays) {
  return renderPlainText(invoice, plays);
}

function renderPlainText(invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

整体流程

初始程序 → 分解函数 → 拆分计算 → 分离文件 → 逻辑重构

- 分离文件的目标是根据逻辑功能将不同逻辑的代码分到对应的文件中。
- 虽然代码的行数增加了，但重构也带来了代码可读性的提高。额外的包装文件将混杂的逻辑分解成可辨别的部分，分离了详单的计算逻辑与样式。

整体流程

初始程序 → 分解函数 → 拆分计算 → 分离文件 → 逻辑重构

- 逻辑重构的目标是根据实际功能需求，引入类型的多态性。

为什么重构

- **重构是一个工具， 它可以用于以下几个目的。**
  - 重构改进软件的设计
  - 重构使软件更容易理解
  - 重构帮助找到bug
  - 重构提高编程速度

何时重构

- **应该把重构融入到编码工作中，其中几个重构的时间点：**
  - 预备性重构： 让添加新功能更容易
  - 帮助理解的重构： 使代码更易懂
  - 捡垃圾式重构
  - 有计划的重构和见机行事的重构
  - 长期重构
  - 复审代码时重构

重构的挑战

- **我认为重构是一种很有价值的技术，大多数团队都应该更多地重构，但它也不是完全没有挑战**
  - 延缓新功能开发
  - 代码所有权
  - 分支
  - 测试
  - 遗留代码
  - 数据库

自动化重构

- **重构已经深入到软件开发的整个工程中，近10年重构领域最大的变化可能就是出现了一批支持自动化重构的工具。**
  - 例如:我想给一个Java的方法改名， 在IntelliJ IDEA或者Eclipse这样的开发环境中， 我只需要从菜单里点选对应的选项， 工具会帮我完成整个重构过程。
  - 重构工具不仅需要理解和修改语法树， 还要知道如何把修改后的代码写回编辑器视图。 总而言之， 实现一个体面的自动化重构手法， 是一个很有挑战的编程任务。

目录

命名

- **整洁代码最重要的一环就是好的名字， 所以我们会深思熟虑如何给函数、 模块、 变量和类命名， 使它们能清晰地表明自己的功能和用法。**
  - 改名可能是最常用的重构手法， 包括改变函数声明 、 变量改名、 字段改名等。

重复代码

- 如果你在一个以上的地点看到相同的代码结构， 那么 设法将它们合而为一， 程序会变得更好。 一旦有重复代码存在， 阅读这些重复的代码时你就必须加倍仔细，留意其间细微的差异。 如果要修改重复代码， 你必须找出所有的副本来修改。
- 最单纯的重复代码就是“同一个类的两个函数含有相同的表达式”。 这时候需要采用提炼函数提炼出重复的代码， 然后让这两个地点都调用被提炼出来的那一段代码。
- 如果重复代码只是相似而不是完全相同， 请首先尝试用移动语句重组代码顺序，把相似的部分放在一起以便提炼。 如果重复的代码段位于同一个超类的不同子类中， 可以使用函数上移来避免在两个子类之间互相调用。

过长函数

- 函数越长， 就越难理解。因此写代码过程中要积极分解函数。 原则： 每当感觉需要以注释来说明的时候， 我们就把需要说明的东西写进一个独立函数中， 并以其用途（而非实现手法） 命名。 我们可以对一组甚至短短一行代码做这件事。哪怕替换后的函数调用动作比函数自身还长， 只要函数名称能够解释其用途， 我们也该毫不犹豫地那么做。
- 关键不在于函数的长度， 而在于函数“做什么”和“如何做”之间的语义距离。

过长参数列表

- 编程过程中，我们应该尽量简化函数的参数列表，常见方式有：
  - 如果可以向某个参数发起查询而获得另一个参数的值， 那么就可以使用以查询取代参数。
  - 如果你发现自己正在从现有的数据结构中抽出很多数据项， 就可以考虑使用保持对象完整手法， 直接传入原来的数据结构。
  - 如果有几项参数总是同时出现， 可以用引入参数对象将其合并成一个对象。
  - 如果某个参数被用作区分函数行为的标记， 可以使用移除标记参数。

全局数据

- 全局数据的问题在于， 从代码库的任何一个角落都可以修改它， 而且没有任何机制可以探测出到底哪段代码做出了修改，常见的修改方法：
  - 封装变量，把全局数据用一个函数包装起来，就能捕捉到修改它的地方， 并开始控制对它的访问。
  - 最好将这个函数（及其封装的数据） 搬移到一个类或模块中， 只允许模块内的代码使用它， 从而尽量控制其作用域。

过大的类

- 不要让单个类做太多事情， 其内往往就会出现太多字段。常见的重构办法：
  - 运用提炼类将几个变量一起提炼至新类内。 提炼时应该选择类内彼此相关的变量， 将它们放在一起。
  - 如果类内的数个变量有着相同的前缀或后缀， 这就意味着有机会把它们提炼到某个组件内。
  - 如果这个组件适合作为一个子类， 提炼超类或者以子类取代类型码（其实就是提炼子类） 往往比较简单。

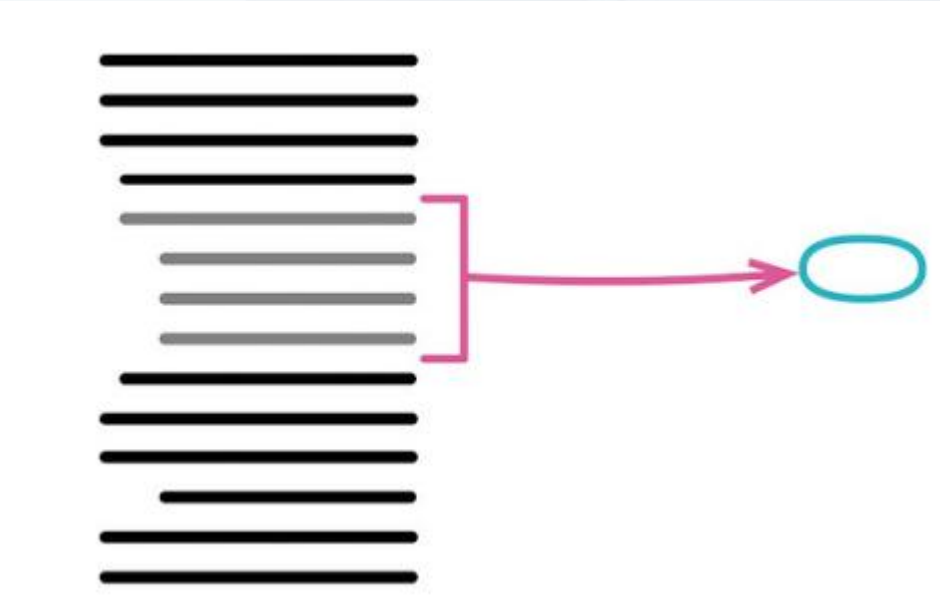
纯数据类

- 纯数据类是指： 它们只拥有一些字段， 以及用于访问（读写） 这些字段的函数。
  他们的编写原则：
  - 避免public字段，应该运用封装记录将它们封装起来。 对于不该被其他类修改的字段， 请运用移除赋值函数。
  - 根据实际需求把处理数据的行为从调用点移动到纯数据类里来。

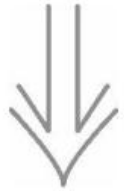
目录

提炼函数

- 浏览代码， 理解其作用， 然后将其提炼到独立的函数中， 并以这段代码的用途为这个函数命名。



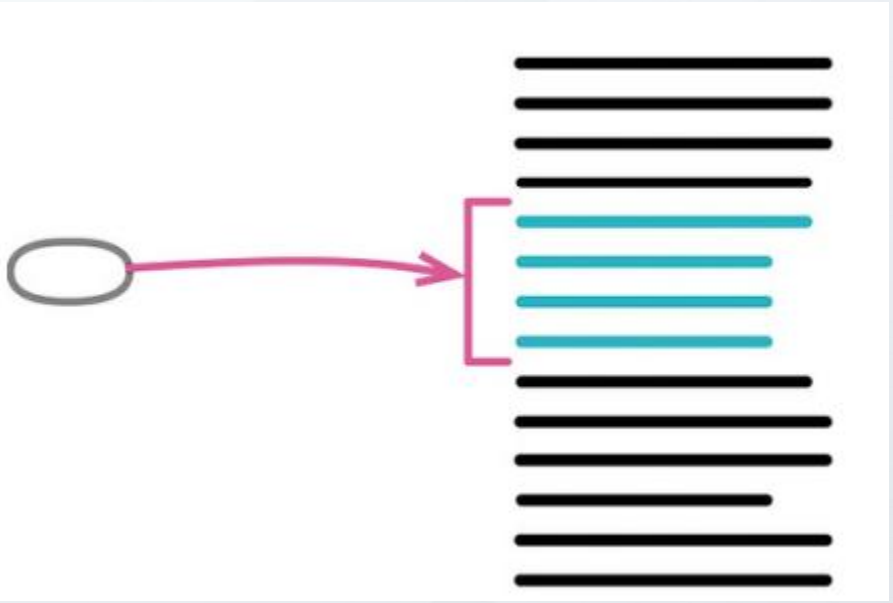
```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```



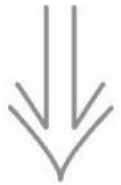
```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();
  printDetails(outstanding);

  function printDetails(outstanding) {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
  }
}
```

内联函数

- 有时候某些函数内部代码和函数名称同样清晰易读。 这时应该去掉这个函数， 直接使用其中的代码。



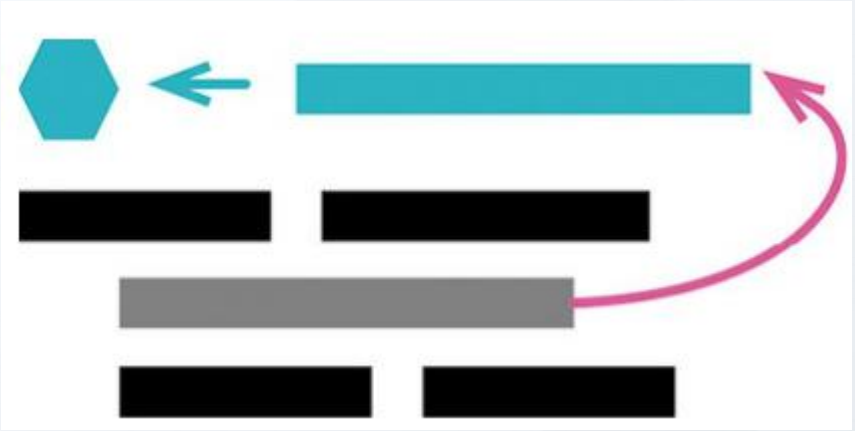
```
function getRating(driver) {
  return moreThanFiveLateDeliveries(driver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(driver) {
  return driver.numberOfLateDeliveries > 5;
}
```



```
function getRating(driver) {
  return (driver.numberOfLateDeliveries > 5) ? 2 : 1;
}
```

提炼变量

- 当表达式非常复杂而难以阅读时，局部变量可以帮助我们将表达式分解为比较容易管理的形式。

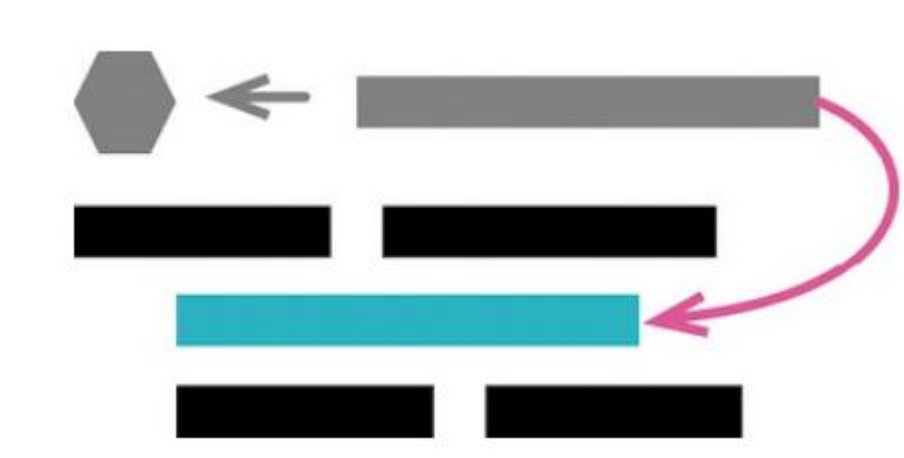


```
return order.quantity * order.itemPrice -
  Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
  Math.min(order.quantity * order.itemPrice * 0.1, 100);
```

```
const basePrice = order.quantity * order.itemPrice;
const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
const shipping = Math.min(basePrice * 0.1, 100);
return basePrice - quantityDiscount + shipping;
```

内联变量

- 在一个函数内部， 但有时候， 变量名字没有表达式本身更具表现力。 还有些时候， 变量可能会妨碍重构附近的代码。 此时就应该通过内联的手法消除变量。

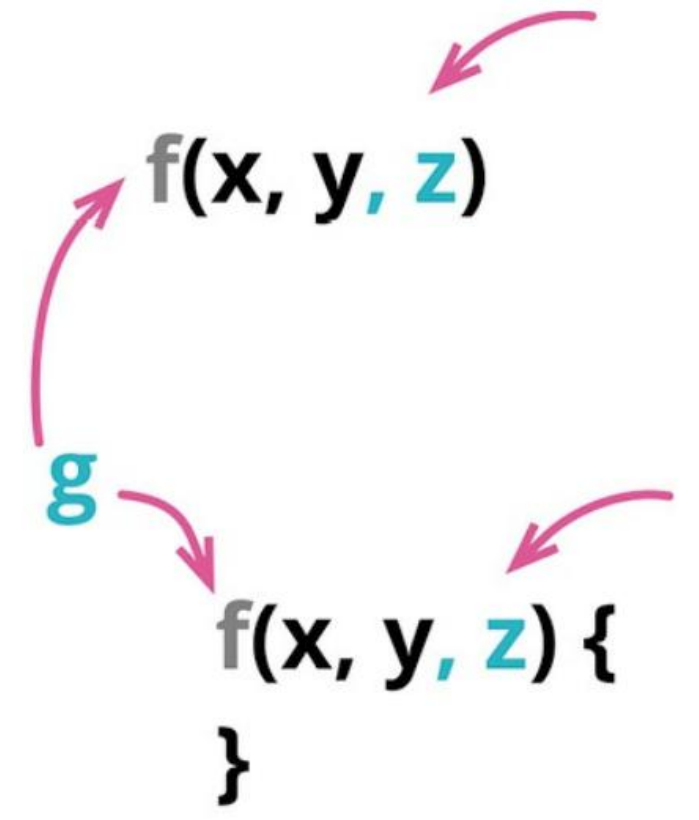


```
let basePrice = anOrder.basePrice;
return (basePrice > 1000);
```

```
return anOrder.basePrice > 1000;
```

改变函数声明

- 函数是我们将程序拆分成小块的主要方式。 函数声明则展现了如何将这些小块组合在一起工作。 系统的好坏很大程度上取决于函数。 可以通过修改函数声明使得系统的结构更加清晰。

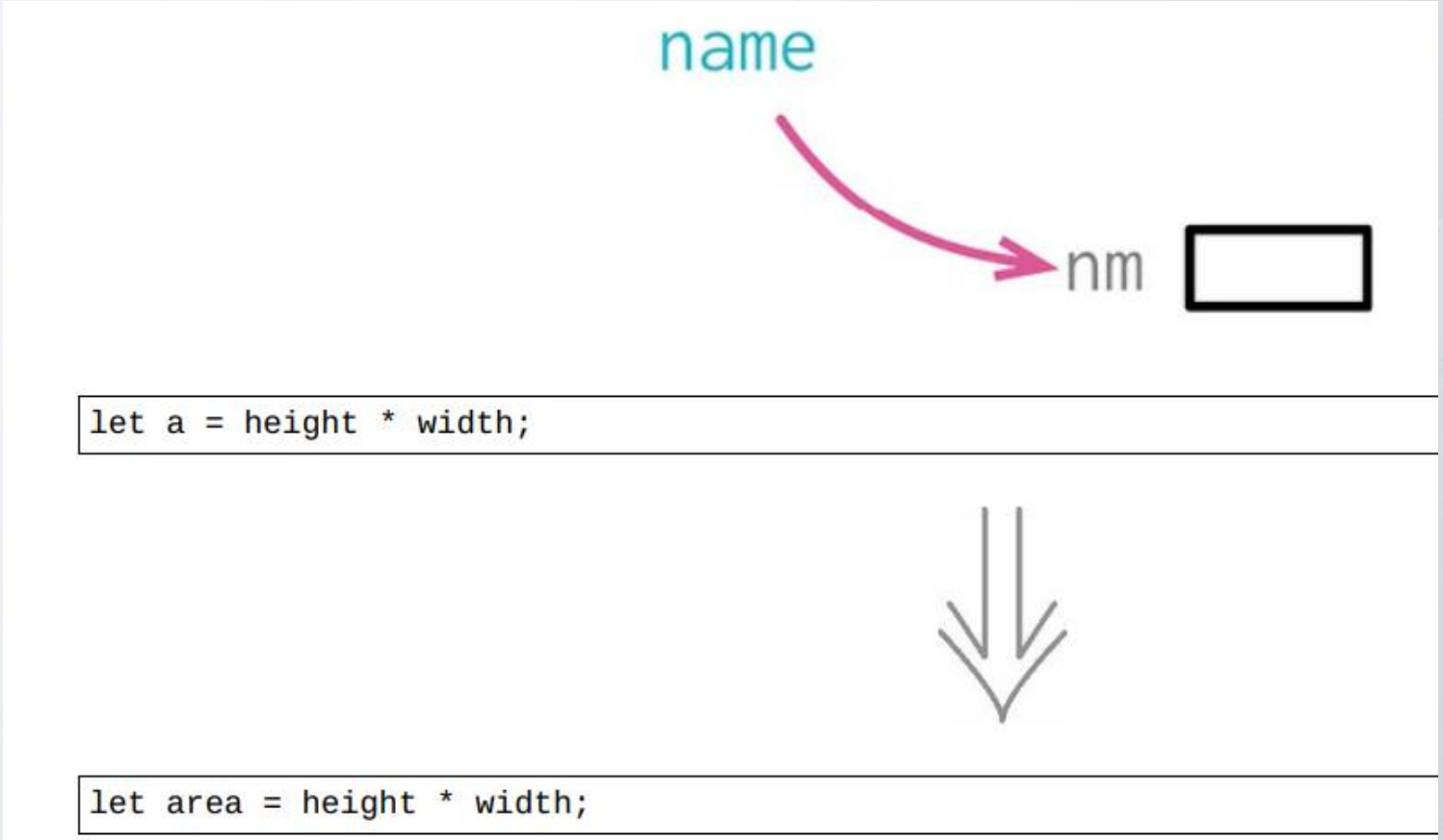



```
function circum(radius) {...}
```

```
function circumference(radius) {...}
```

变量改名

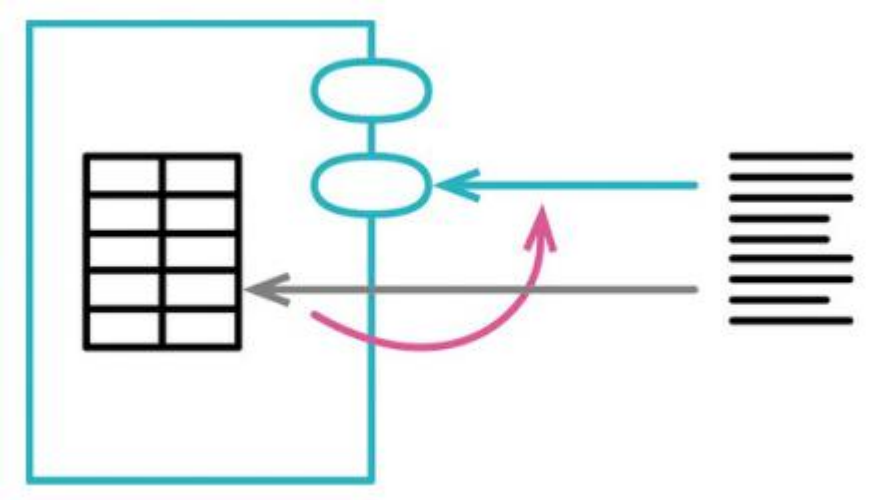
- 好的命名是整洁编程的核心。 好的变量名可以很好地解释一段程序的功能。



目录

封装记录

- 应该使用类对象而非记录型数据 。

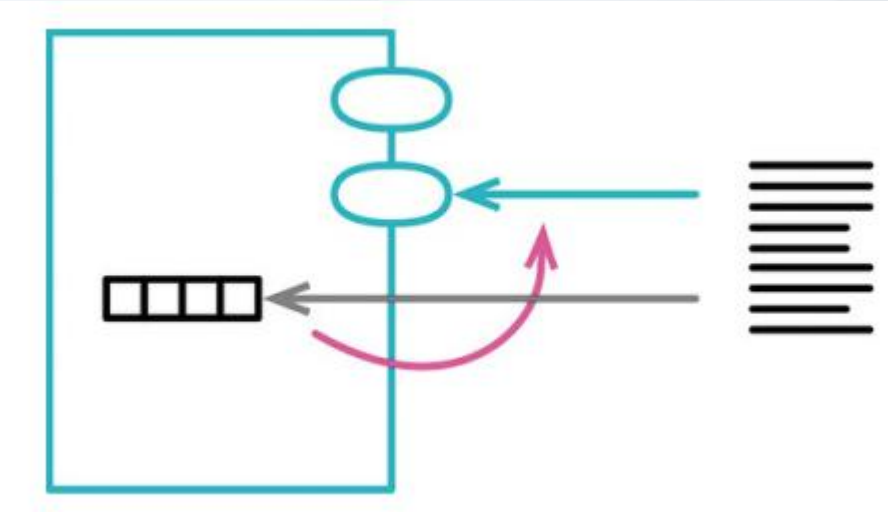


```
organization = {name: "Acme Gooseberries", country: "GB"};
```

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(arg) {this._name = arg;}
  get country()    {return this._country;}
  set country(arg) {this._country = arg;}
}
```

封装集合

- 封装集合时人们常常犯一个错误： 只对集合变量的访问进行了封装， 但依然让取值函数返回集合本身。 这使得集合的成员变量可以直接被修改。
- 应该增加 “添加” 和 “移除” 方法。 这样可使对集合的修改必须经过类， 当程序出现错误时，能轻易找出修改点。

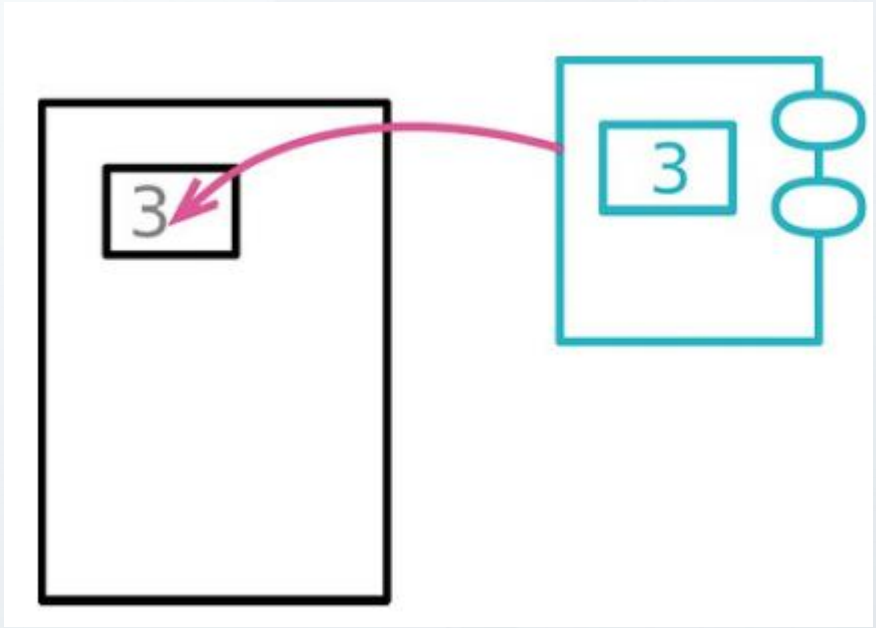


```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

```
class Person {
  get courses() {return this._courses.slice();}
  addCourse(aCourse) { ... }
  removeCourse(aCourse) { ... }
```

以对象取代级别类型

- 某个数据的操作不仅仅局限于打印时， 应为它创建一个新类。 一开始这个类也许只是简单包装一下简单类型的数据， 不过日后添加的业务逻辑就可以方便加入。

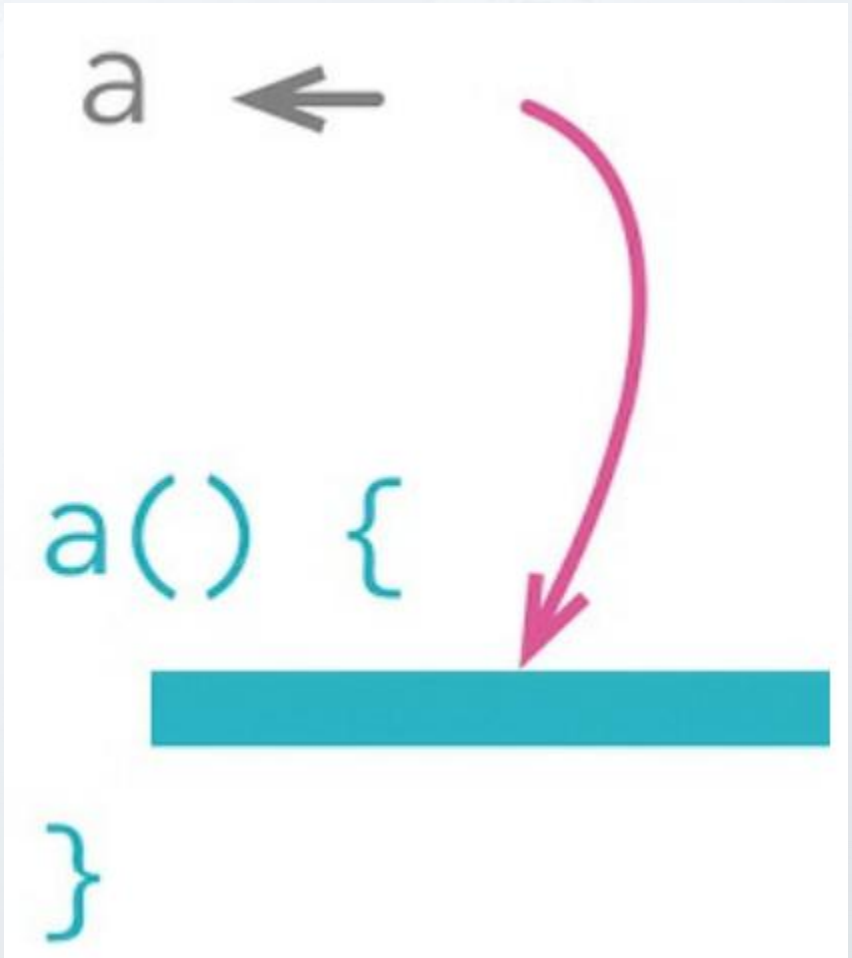



```
orders.filter(o => "high" === o.priority
                || "rush" === o.priority);
```

```
orders.filter(o => o.priority.higherThan(new Priority("normal")))
```

以查询取代临时变量

- 临时变量的作用是保存某段代码的返回值，很多时候需要将它们抽取成函数




```
const basePrice = this._quantity * this._itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

```
get basePrice() {this._quantity * this._itemPrice;}

...

if (this.basePrice > 1000)
  return this.basePrice * 0.95;
else
  return this.basePrice * 0.98;
```

目录

总结

1. 重构介绍
2. 代码重构
3. 函数重构
4. 类级别重构
5. 总结

THANKS