Refactoring: A First Example

Martin Fowler’s First Example of Refactoring, Adapted to Scala

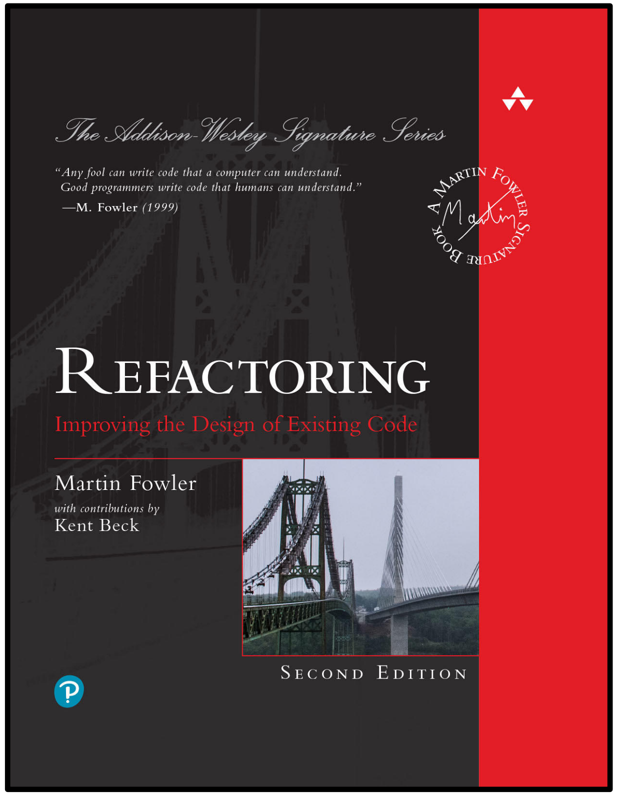
follow in the footsteps of **refactoring guru Martin Fowler**

as he **improves** the **design** of a program in a simple yet **instructive refactoring example**

whose **JavaScript** code and associated **refactoring** is herein adapted to **Scala**

based on the second edition of ‘the’ **Refactoring** book

Martin Fowler
@martinfowler



The Addison-Wesley Signature Series

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
—M. Fowler (1999)

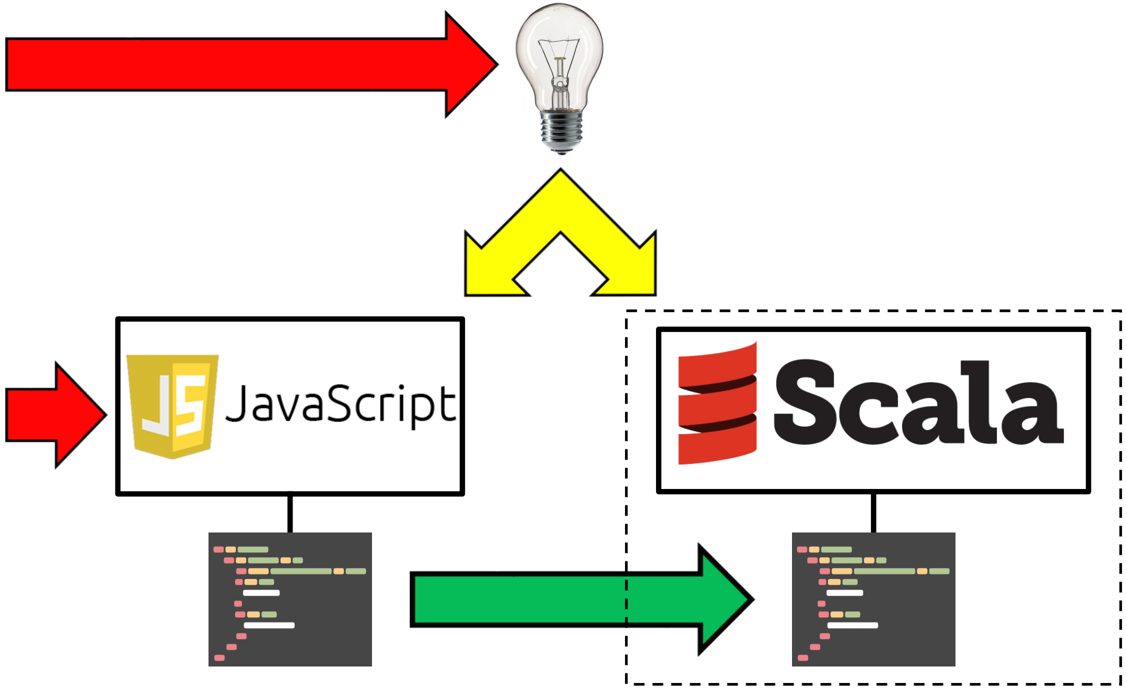
A Martin Fowler Signature Book

REFACTORING

Improving the Design of Existing Code

Martin Fowler
with contributions by
Kent Beck

SECOND EDITION



slides by

@philip_schwarz slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

Neither **Martin Fowler** nor the **Refactoring** book need any introduction.

I have always been a great fan of both, and having finally found the time to study in detail the **refactoring example** in the **second edition** of the book, I would like to share the experience of adapting to **Scala** such a useful **example**, which happens to be written in **JavaScript**.

Another reason for looking in detail at the **example** is that it can be used as a good **refactoring code kata**.

While we'll be closely following **Martin Fowler**'s footsteps as he works through the **refactoring example**, and while those of you who don't already own a copy of the book, will no doubt learn a lot about the chapter containing the **example**, what we'll see is obviously only a small part of what makes the book such a must have for anyone interested in **refactoring**.

The next four slides consist of excerpts in which **Martin Fowler** introduces the program whose **design** he will be **improving** through **refactoring**.

So I'm going to start this book with an example of refactoring. I'll talk about how refactoring works and will give you a sense of the refactoring process. I can then do the usual principles-style introduction in the next chapter.

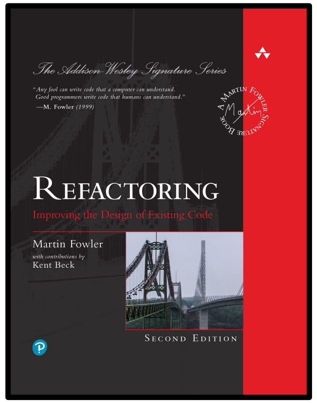
With any introductory example, however, I run into a problem. If I pick a large program, describing it and how it is refactored is too complicated for a mortal reader to work through. (I tried this with the original book—and ended up throwing away two examples, which were still pretty small but took over a hundred pages each to describe.) **However, if I pick a program that is small enough to be comprehensible, refactoring does not look like it is worthwhile**.

I'm thus in the classic bind of anyone who wants to describe techniques that are useful for real-world programs.

Frankly, it is not worth the effort to do all the refactoring that I'm going to show you on the small program I will be using.

But if the code I'm showing you is part of a larger system, then the refactoring becomes important. Just look at my example and imagine it in the context of a much larger system.

Martin Fowler
@martinfowler


The Addison Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by Kent Beck
Second Edition

I chose JavaScript to illustrate these refactorings, as I felt that this language would be readable by the most amount of people.

JavaScript

You shouldn't find it difficult, however, to adapt the refactorings to whatever language you are currently using.

I try not to use any of the more complicated bits of the language, so you should be able to follow the refactorings with only a cursory knowledge of JavaScript.

My use of JavaScript is certainly not an endorsement of the language.

Although I use JavaScript for my examples, that doesn't mean the techniques in this book are confined to JavaScript.

The first edition of this book used Java, and many programmers found it useful even though they never wrote a single Java class.

I did toy with illustrating this generality by using a dozen different languages for the examples, but I felt that would be too confusing for the reader.

Still, this book is written for programmers in any language.

Outside of the example sections, I'm not making any assumptions about the language.

I expect the reader to absorb my general comments and apply them to the language they are using.

Indeed, I expect readers to take the JavaScript examples and adapt them to their language.

Martin Fowler

@martinfowler

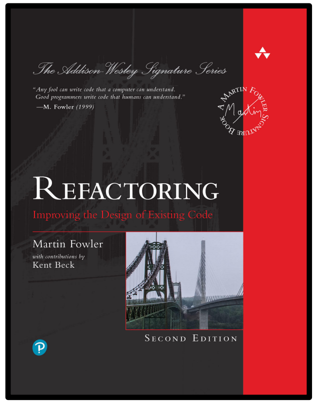

The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by Kent Beck
SECOND EDITION

Image a company of theatrical players who go out to various events performing plays.

Typically, a customer will request a few plays and the company charges them based on the size of the audience and the kind of play they perform.

There are currently two kinds of plays that the company performs: tragedies and comedies.

As well as providing a bill for the performance, the company gives its customers “volume credits” which they can use for discounts on future performances—think of it as a customer loyalty mechanism.

The performers store data about their plays in a simple JSON file that looks something like this:

```
plays.json…

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

The data for their bills also comes in a JSON file:

```
invoices.json…

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

```
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;
}
```

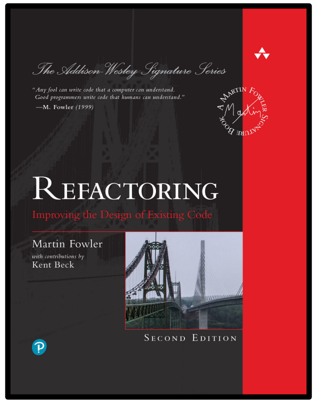
The code that prints the bill is this simple function.

What are your thoughts on the design of this program? The first thing I'd say is that it's tolerable as it is—a program so short doesn't require any **deep structure** to be **comprehensible**. But remember my earlier point that I have to keep examples small. **Imagine this program on a larger scale—perhaps hundreds of lines long. At that size, a single inline function is hard to understand.**

Given that the program works, isn't any statement about its structure merely an aesthetic judgment, a dislike of "ugly" code? After all, the compiler doesn't care whether the code is ugly or clean. But when I change the system, there is a human involved, and humans do care. A poorly designed system is hard to change—because it is difficult to figure out what to change and how these changes will interact with the existing code to get the behavior I want. And if it is hard to figure out what to change, there is a good chance that I will make mistakes and introduce bugs.

Thus, **if I'm faced with modifying a program with hundreds of lines of code, I'd rather it be structured into a set of functions and other program elements that allow me to understand more easily what the program is doing. If the program lacks structure, it's usually easier for me to add structure to the program first, and then make the change I need.**

Martin Fowler
@martinfowler


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by Kent Beck
Second Edition

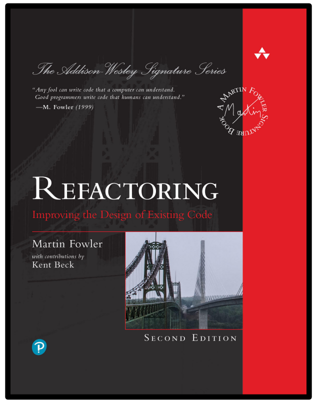
```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD", minimumFractionDigits: 2 }).format;

  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;
}
```

In this case, I have a couple of changes that the users would like to make. First, they want a statement printed in HTML. Consider what impact this change would have. I'm faced with adding conditional statements around every statement that adds a string to the result. That will add a host of complexity to the function. Faced with that, most people prefer to copy the method and change it to emit **HTML**. Making a copy may not seem too onerous a task, but it sets up all sorts of problems for the future. Any changes to the charging logic would force me to update both methods—and to ensure they are updated consistently. **If I'm writing a program that will never change again, this kind of copy-and-paste is fine. But if it's a long-lived program, then duplication is a menace.**

This brings me to a second change. The players are looking to perform more kinds of plays: they hope to add **history**, **pastoral**, **pastoral-comical**, **historical-pastoral**, **tragical-historical**, **tragical-comical-historical-pastoral**, **scene individable**, and **poem unlimited** to their repertoire. They haven't exactly decided yet what they want to do and when. **This change will affect both the way their plays are charged for and the way volume credits are calculated**. As an experienced developer I can be sure that whatever scheme they come up with, they will change it again within six months. After all, when feature requests come, they come not as single spies but in battalions.

Martin Fowler

@martinfowler


The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by Kent Beck
Second Edition

@philip_schwarz

In this slide deck we are going to

1. Translate **Martin Fowler**'s initial **Javascript** program into **Scala**
2. Follow in his **refactoring** footsteps, transforming our **Scala** program so that it is **easier** to **understand** and **easier** to **change**.

On the very few occasions when a decision is made that turns out not to be a good fit in a **Scala** context, we'll make an alternative decision that is more suitable for the **Scala** version of the program.

To keep the pace snappy, we'll sometimes coalesce a few of **Martin**'s **refactoring nanosteps** or **microsteps** into one (see next slide for a definition of these two types of **refactoring** step).

J. B. Rainsberger
@jbrains

Some Helpful Terms

In my lexicon, **a *nanostep* is something like adding a new field to a class**. Another **nanostep** is finding code that wrote to an existing field and adding code that writes the corresponding value to the new field, keeping their values synchronized with each other. Yet another is remembering the keystroke for “extract variable” so that you can simply type the expression (right-hand value) that you have in mind first, then assign it to a new variable (and let the computer compute the type of the variable for you).

A *microstep* is a collection of related nanosteps like introducing an interface *and* changing a few classes to implement that interface, adding empty/default method implementations to the classes that now need it. Another is pushing a value up out of the constructor into its parameter list. Yet another is remembering that you can either extract a value to a variable before extracting code into a method or you can extract the method first, then introduce the value as a parameter, and which keystrokes in NetBeans make that happen.

A *move* is a collection of related microsteps, like inverting the dependency between A and B, where A used to invoke B, but now A fires an event which B subscribes to and handles.

https://blog.thecodewhisperer.com/permalink/breaking-through-your-refactoring-rut

Let’s knock up some **Scala** data structures for **plays**, **invoices** and **performances**.

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

```
val plays: Map[String, Play] = Map (
  "hamlet"  -> Play(name = "Hamlet", `type` = "tragedy"),
  "as-like" -> Play(name = "As You Like It", `type` = "comedy"),
  "othello" -> Play(name = "Othello", `type` = "tragedy")
)
```

```
case class Play(name: String, `type`: String)
```

```
case class Invoice(customer: String, performances: List[Performance])
```

```
case class Performance(playID: String, audience: Int)
```

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

```
val invoices: List[Invoice] = List(
  Invoice( customer = "BigCo",
           performances = List(
             Performance(playID = "hamlet",
                         audience = 55),
             Performance(playID = "as-like",
                         audience = 35),
             Performance(playID = "othello",
                         audience = 40)))
)
```

```javascript
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;
}
```

JavaScript

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match

      case "tragedy" =>

        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)

      case "comedy" =>

        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience

      case other =>

        throw IllegalArgumentException(s"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` then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Scala

Here is a literal translation of the **Javascript** program into **Scala**.

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match

      case "tragedy" =>
        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)

      case "comedy" =>
        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience

      case other =>
        throw IllegalArgumentException(s"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` then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Here is the **Scala** code again, together with the data structures we created earlier, and also a simple **regression test** consisting of a single **assertion**.

```scala
case class Performance(playID: String, audience: Int)

case class Invoice(customer: String, performances: List[Performance])

case class Play(name: String, `type`: String)
```

```scala
val invoices: List[Invoice] = List(
  Invoice( customer = "BigCo",
           performances = List(Performance(playID = "hamlet",
                                           audience = 55),
                               Performance(playID = "as-like",
                                           audience = 35),
                               Performance(playID = "othello",
                                           audience = 40)))
)

val plays = Map (
  "hamlet"  -> Play(name = "Hamlet", `type` = "tragedy"),
  "as-like" -> Play(name = "As You Like It", `type` = "comedy"),
  "othello" -> Play(name = "Othello", `type` = "tragedy")
)
```

```scala
@main def main: Unit =
  assert(
    statement(invoices(0), plays)
    ==
    """|Statement for BigCo
       |  Hamlet: $650.00 (55 seats)
       |  As You Like It: $580.00 (35 seats)
       |  Othello: $500.00 (40 seats)
       |Amount owed is $1,730.00
       |You earned 47 credits
       |""".stripMargin
  )
```

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match

      case "tragedy" =>
        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)

      case "comedy" =>
        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience

      case other =>
        throw IllegalArgumentException(s"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` then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Yes, I hear you! Using **mutable variables** is very uncommon in **Scala**.

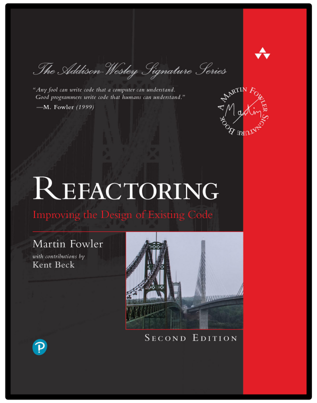
We are only using such variables in order to be faithful to **Martin Fowler**'s initial **Javascript** program.

Don't worry: as we refactor the code, we'll slowly but surely eliminate such **mutability**.

Martin Fowler

@martinfowler

Decomposing the `statement` Function


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match

      case "tragedy" =>
        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)

      case "comedy" =>
        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience

      case other =>
        throw IllegalArgumentException(s"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` then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

When refactoring a long function like this, I mentally try to identify points that separate different parts of the overall behaviour.

The first chunk that leaps to my eye is the switch statement in the middle.

Martin Fowler

@martinfowler

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match

      case "tragedy" =>

        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)

      case "comedy" =>

        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience

      case other =>

        throw IllegalArgumentException(s"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`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = amountFor(perf,play)

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

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

- **Extract Function amountFor**
- In **amountFor** function:
  - rename **perf** arg to **aPerformance**
  - rename **thisAmount** arg to **result**

```
def amountFor(aPerformance: Performance, play: Play): Int =
  var result = 0
  play.`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw IllegalArgumentException(s"unknown type ${play.`type`}")
  result
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def amountFor(aPerformance: Performance, play: Play): Int =
    var result = 0
    playFor(perf).`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw new IllegalArgumentException(s"unknown type ${playFor(perf).`type`}")
    result

  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = amountFor(perf,play)

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

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

It makes sense for subordinate functions extracted from the **statement** function to be nested inside it.

However, in the interest of clarity and brevity, I will at times show the **statement** function without also showing such subordinate functions.

In the previous slide for example, although the **amountFor** function was extracted from **statement**, it is shown outside **statement** rather than nested inside it.

In the **statement** function on the left however, we do see **amountFor** nested inside **statement**.

@philip_schwarz

Martin Fowler

@martinfowler

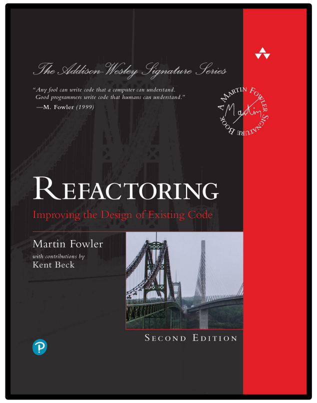
The next item to consider for renaming is the **play parameter**, but I have a **different fate** for that.

```
def amountFor(aPerformance: Performance, play: Play): Int =
  var result = 0
  play.`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw new IllegalArgumentException(s"unknown type ${play.`type`}")
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable


The Addison Wesley Signature Series
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Removing the play Variable

The next two slides perform a **Replace Temp with Query** **refactoring** on the **play** variable.

Such a **refactoring** is itself composed of the following **refactorings**:

- **Extract Function**
- **Inline Variable**

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = amountFor(perf,play)

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

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Removing the play Variable

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = amountFor(perf,play)

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

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

- **Extract Function playFor**
- rename **playFor perf** parameter to **aPerformance**

```
def playFor(aPerformance: Performance): Play =
  plays(aPerformance.playID)
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = playFor(perf)
    var thisAmount = amountFor(perf,play)

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

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Removing the play Variable

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = playFor(perf)
    var thisAmount = amountFor(perf,play)

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

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Inline Variable play in **statement** function

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    var thisAmount = amountFor(perf,playFor(perf))

    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Removing the play Variable

in **amountFor** function: replace references to **play** parameter with invocations of **playFor** function

```
def amountFor(aPerformance: Performance, play: Play): Int =
  var result = 0
  play.`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw IllegalArgumentException(s"unknown type ${play.`type`}")
  result
```

```
def amountFor(aPerformance: Performance, play: Play): Int =
  var result = 0
  playFor(aPerformance).`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw IllegalArgumentException(s"unknown type ${playFor(aPerformance).`type`}")
  result
```

Removing the play Variable

```
def amountFor(aPerformance: Performance, play: Play): Int =
  var result = 0
  playFor(perf).`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw new IllegalArgumentException(s"unknown type ${playFor(perf).`type
  result
```

```
def amountFor(aPerformance: Performance): Int =
  var result = 0
  playFor(perf).`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw new IllegalArgumentException(s"unknown type ${playFor(perf).`type`}")
  result
```

Change Function Declaration of **amountFor** by removing **play** parameter

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    var thisAmount = amountFor(perf,playFor(perf))

    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(thisAmount/100)}
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    var thisAmount = amountFor(perf)

    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

Now that I am done with the arguments to **amountFor**, I look back at where it’s called.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    var thisAmount = amountFor(perf)

    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    var thisAmount = amountFor(perf)

    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

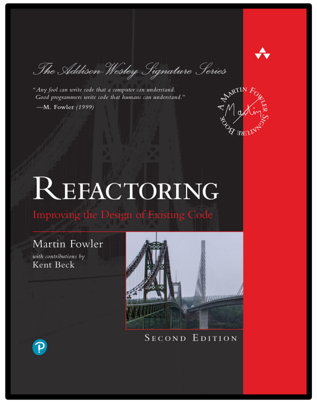
Inline Variable thisAmount in statement function

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits


The Addison Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Extracting Volume Credits

Martin Fowler

@martinfowler

Now I get the **benefit** from removing the **play variable** as it makes it easier to extract the **volume credits** calculation by removing one of the locally scoped variables. I still have to deal with the other two.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Extracting Volume Credits

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    // add volume credits
    volumeCredits += math.max(perf.audience - 30, 0)
    // add extra credit for every ten comedy attendees
    if "comedy" == playFor(perf).`type`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

- **Extract Function volumeCreditsFor**
- In **volumeCreditsFor** function:
  - rename **perf** arg to **aPerformance**
  - rename **volumeCredits** arg to **result**

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

```
def volumeCreditsFor(aPerformance: Performance): Int =
  var result = 0
  result += math.max(aPerformance.audience - 30, 0)
  if "comedy" == playFor(aPerformance).`type` then result += math.floor(aPerformance.audience / 5).toInt
  result
```

Martin Fowler
@martinfowler

As I suggested before, **temporary variables** can be a problem. They are only useful within their own routine, and therefore encourage **long**, **complex routines**.

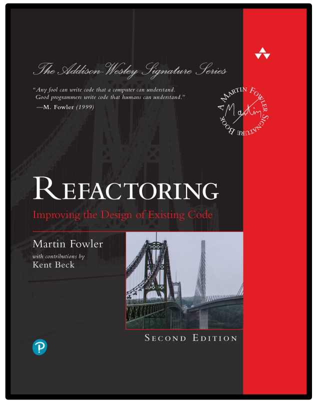
My next move, then, is to replace some of them. The easiest one is **formatter**.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Removing the `formatter` Variable

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${formatter.format(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

- **Extract Function format**
- Replace references to **formatter.format** with invocations of **format**
- **Change Function Declaration** of **format** by renaming function to **usd**

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

```scala
def usd(aNumber: Int): String =
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))
  formatter.format(aNumber)
```

Martin Fowler
@martinfowler

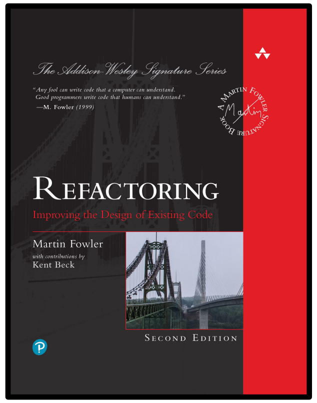
My next terget variable is **volumeCredits**. This is a trickier case, as it’s built up during the iterations of the loop.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Removing Total Volume Credits

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)
  end for

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

- Apply **Split Loop** to the loop on invoice.performances
- Apply **Slide Statements** to the statement initialising variable volumeCredits

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0

  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)

  var volumeCredits = 0
  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Removing Total Volume Credits

The next two slides perform a **Replace Temp with Query refactoring** on the **volumeCredits** variable.

As we saw earlier on, such a **refactoring** is itself composed of the following **refactorings**:

- **Extract Function**
- **Inline Variable**

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0

  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)

  var volumeCredits = 0
  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

Removing Total Volume Credits

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0

  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)

  var volumeCredits = 0
  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

- Extract Function totalVolumeCredits
- Inline Variable volumeCredits

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0

  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```scala
def totalVolumeCredits: Int =
  var volumeCredits = 0
  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)
  volumeCredits
```

Martin Fowler
@martinfowler

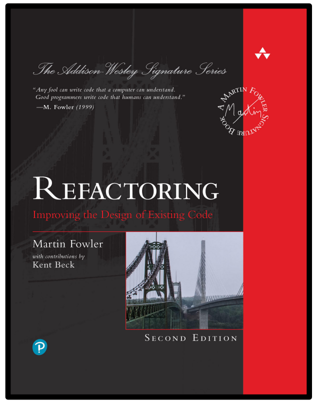
I then repeat that sequence to remove totalAmount.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0

  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Removing Total Amount

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0

  var result = s"Statement for ${invoice.customer}\n"

  for (perf <- invoice.performances)

    // print line for this order
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
    totalAmount += amountFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

- Apply **Split Loop** to the loop on invoice.performances
- Apply **Slide Statements** to the statement initialising variable totalAmount

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"

  var totalAmount = 0
  for (perf <- invoice.performances)
    totalAmount += amountFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Removing Total Amount

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"

  var totalAmount = 0
  for (perf <- invoice.performances)
    totalAmount += amountFor(perf)

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

- **Extract Function appleSauce**
- **Inline Variable totalAmount**

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"

  result += s"Amount owed is ${usd(appleSauce/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
def appleSauce: Int =
  var totalAmount = 0
  for (perf <- invoice.performances)
    totalAmount += amountFor(perf)
  totalAmount
```

Removing Total Amount

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"

  result += s"Amount owed is ${usd(appleSauce/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```scala
def totalVolumeCredits: Int =
  var volumeCredits = 0
  for (perf <- invoice.performances)
    volumeCredits += volumeCreditsFor(perf)
  volumeCredits
```

```scala
def appleSauce: Int =
  var totalAmount = 0
  for (perf <- invoice.performances)
    totalAmount += amountFor(perf)
  totalAmount
```

- **Change Function Declaration** of **appleSauce** by renaming function to **totalAmount**
- **Rename Variables** **volumeCredits** and **totalAmount** to **result**

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"

  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

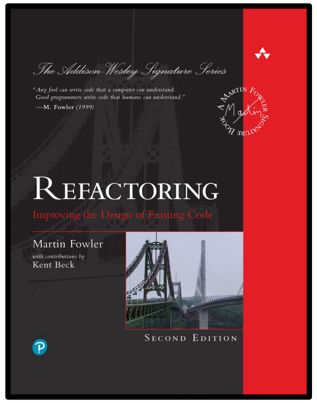
```scala
def totalVolumeCredits: Int =
  var result = 0
  for (perf <- invoice.performances)
    result += volumeCreditsFor(perf)
  result
```

```scala
def totalAmount: Int =
  var result = 0
  for (perf <- invoice.performances)
    result += amountFor(perf)
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions



Status: Lots of Nested Functions

Now is a good time to pause and take a look at the overall state of the code.

The structure of the code is much better now.

The top-level statement function is now just six lines of code, and all it does is laying out the printing of the statement.

All the calculation logic has been moved out to a handful of supporting functions.

This makes it easier to understand each individual calculation as well as the overall flow of the report.

Martin Fowler
@martinfowler

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += amountFor(perf)
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += volumeCreditsFor(perf)
    result

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  def volumeCreditsFor(aPerformance: Performance): Int =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type` then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance) =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Original Program

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match
      case "tragedy" =>
        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)
      case "comedy" =>
        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience
      case other =>
        throw IllegalArgumentException(s"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` then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

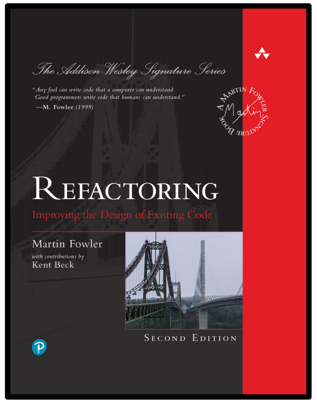
Refactored Program

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += amountFor(perf)
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += volumeCreditsFor(perf)
    result

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  def volumeCreditsFor(aPerformance: Performance): Int =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type` then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Splitting the Phases of Calculation and Formatting

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += amountFor(perf)
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += volumeCreditsFor(perf)
    result

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  def volumeCreditsFor(aPerformance: Performance): Int =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

So far, my refactoring has focused on adding enough structure to the function so that I can understand it and see it in terms of its logical parts.

This is often the case early in refactoring. Breaking down complicated chunks into small pieces is important, as is naming things well.

Now, I can begin to focus more on the functionality change I want to make—specifically, providing an HTML version of this statement.

In many ways, it's now much easier to do. With all the calculation code split out, all I have to do is write an HTML version of the six lines of code at the bottom.

The problem is that these broken-out functions are nested within the textual statement method, and I don't want to copy and paste them into a new function, however well organized.

Martin Fowler

@martinfowler

Splitting the Phases of Calculation and Formatting

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += amountFor(perf)
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += volumeCreditsFor(perf)
    result

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  def volumeCreditsFor(aPerformance: Performance): Int =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

I want the same calculation functions to be used by the text and HTML versions of the statement.

There are various ways to do this, but one of my favorite techniques is Split Phase.

My aim here is to divide the logic into two parts: one that calculates the data required for the statement, the other that renders it into text or HTML.

The first phase creates an intermediate data structure that it passes to the second.

I start a Split Phase by applying Extract Function to the code that makes up the second phase.

In this case, that's the statement printing code, which is in fact the entire content of statement.

This, together with all the nested functions, goes into its own top-level function which I call renderPlainText (see next slide).

Martin Fowler

@martinfowler

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += amountFor(perf)
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += volumeCreditsFor(perf)
    result

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Extract Function renderPlainText

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  renderPlainText(invoice, plays)

def renderPlainText(invoice: Invoice, plays: Map[String, Play]): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += amountFor(perf)
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- invoice.performances)
      result += volumeCreditsFor(perf)
    result

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

In upcoming slides, **Martin Fowler** will be using the concept of a **Javascript Object**, which he creates and then adds fields to:

```
const foo = {};
foo.bar = abc;
foo.baz = def
```

What we'll be doing instead in **Scala** is introduce a **case class**:

```
case class Foo(bar: Bar, baz: Baz)
```

Splitting the Phases of Calculation and Formatting

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  renderPlainText(invoice, plays)

def renderPlainText(invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Martin Fowler
@martinfowler

I do my usual compile-test-commit, then create an object that will act as my intermediate data structure between the two phases. I pass this data object in as an argument to renderPlainText.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val statementData = StatementData()
  renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class StatementData()
```

Splitting the Phases of Calculation and Formatting

```scala
case class StatementData()
```

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val statementData = StatementData()
  renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${invoice.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Martin Fowler

@martinfowler

I now examine the other arguments used by renderPlainText. I want to move the data that comes from them into the intermediate data structure, so that all the calculation code moves into the statement function and renderPlainText operates solely on data passed to it through the data parameter.

My first move is to take the customer and add it to the intermediate object.

```scala
case class StatementData(customer: String)
```

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val statementData = StatementData(invoice.customer)
  renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${data.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Splitting the Phases of Calculation and Formatting

```
case class StatementData(customer: String)
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val statementData = StatementData(invoice.customer)
  renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, invoice: Invoice, plays: Map[String, Play]): String =
  var result = s"Statement for ${data.customer}\n"
  for (perf <- invoice.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
def totalVolumeCredits: Int =
  var result = 0
  for (perf <- invoice.performances)
    result += volumeCreditsFor(perf)
  result
```

```
def totalAmount: Int =
  var result = 0
  for (perf <- invoice.performances)
    result += amountFor(perf)
  result
```

Similarly, I add the performances, which allows me to delete the invoice parameter to renderPlainText.

Martin Fowler
@martinfowler

```
case class StatementData(customer: String, performances: List[Performance])
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val statementData = StatementData(invoice.customer, invoice.performances)
  renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, plays: Map[String, Play]): String =
  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result += s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
def totalVolumeCredits: Int =
  var result = 0
  for (perf <- data.performances)
    result += volumeCreditsFor(perf)
  result
```

```
def totalAmount: Int =
  var result = 0
  for (perf <- data.performances)
    result += amountFor(perf)
  result
```

@philip_schwarz

In upcoming slides, Martin Fowler introduces the notion of **'enriching'** **Performance** objects (during the **calculation phase**) with additional fields (that are to be used during the **formatting phase**).

Whilst in **Scala** we'll ultimately aim to have both a **Performance case class** and an **EnrichedPerformance case class**, we'll have to start off by **'enriching'** the **Performance case class** with **optional fields**, and only later remove the **optional fields** in favour of a new **EnrichedPerformance case class**.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val statementData = StatementData(invoice.customer, invoice.performances)
  renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, plays: Map[String, Play]): String =
  def amountFor(aPerformance: Performance): Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def playFor(aPerformance: Performance) =
    plays(aPerformance.playID)

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${playFor(perf).name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(playID: String, audience: Int)
```

Now I'd like the play name to come from the intermediate data. To do this, I need to enrich the performance record with data from the play.

Martin Fowler @martinfowler

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): Performance =
    Performance(aPerformance.playID, Some(playFor(aPerformance)), aPerformance.audience)

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
    renderPlainText(statementData, invoice, plays)

def renderPlainText(data: StatementData, plays: Map[String, Play]): String =
  def amountFor(aPerformance: Performance): Int =
    var result = 0
    aPerformance.play.get.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${aPerformance.play.get.`type`}")
    result

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == aPerformance.play.get.`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.get.name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(playID: String, play: Option[Play] = None ,audience: Int)
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): Performance =
    Performance(
      aPerformance.playID,
      Some(playFor(aPerformance)),
      aPerformance.audience)

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData, plays)

def renderPlainText(data: StatementData, plays: Map[String, Play]): String =
  def amountFor(aPerformance: Performance): Int =
    var result = 0
    aPerformance.play.get.`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw new IllegalArgumentException(s"unknown type ${aPerformance.play.get.`type`}")
    result

  def totalAmount: Int =
    var result = 0
    for (perf <- data.performances)
      result += amountFor(perf)
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.get name}: ${usd(amountFor(perf)/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(
  playID: String,
  play: Option[Play] = None,
  audience: Int)
```

Martin Fowler

I then move amountFor in a similar way.

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): Performance =
    Performance(
      aPerformance.playID,
      Some(playFor(aPerformance)),
      aPerformance.audience,
      Some(amountFor(aPerformance)))

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    playFor(aPerformance).`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${playFor(aPerformance).`type`}")
    result

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  def totalAmount: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.amount.get
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.get name}: ${usd(perf.amount.get/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(
  playID: String,
  play: Option[Play] = None,
  audience: Int,
  amount: Option[Int] = None)
```

Note that, on the previous slide, I have already removed the **plays** parameter of **renderPlainText**, since it is no longer used. In the book, this doesn't happen till later in this section.

Splitting the Phases of Calculation and Formatting

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): Performance =
    Performance(
      aPerformance.playID,
      Some(playFor(aPerformance)),
      aPerformance.audience,
      Some(amountFor(aPerformance)))

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == aPerformance.play.get.`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- data.performances)
      result += volumeCreditsFor(perf)
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.get name}: ${usd(perf.amount.get/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Next, I move the volumeCreditsFor calculation.

Martin Fowler
@martinfowler

```
case class Performance(
  playID: String,
  play: Option[Play] = None,
  audience: Int,
  amount: Option[Int] = None)
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): Performance =
    Performance(
      aPerformance.playID,
      Some(playFor(aPerformance)),
      aPerformance.audience,
      Some(amountFor(aPerformance)),
      Some(volumeCreditsFor(aPerformance)))

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.volumeCredits.get
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.get name}: ${usd(perf.amount.get/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(
  playID: String,
  play: Option[Play] = None,
  audience: Int,
  amount: Option[Int] = None,
  volumeCredits: Option[Int] = None)
```

We can now remove the **optional** **Performance** fields by introducing an **EnrichedPerformance**.

Splitting the Phases of Calculation and Formatting

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): Performance =
    Performance(
      aPerformance.playID,
      Some(playFor(aPerformance)),
      aPerformance.audience,
      Some(amountFor(aPerformance)),
      Some(volumeCreditsFor(aPerformance)))

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.volumeCredits.get
    result

  def totalAmount: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.amount.get
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.get.name}: ${usd(perf.amount.get/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(
  playID: String,
  play: Option[Play] = None,
  audience: Int,
  amount: Option[Int] = None,
  volumeCredits: Option[Int] = None)
```

```
case class StatementData(customer: String, performances: List[Performance])
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
    EnrichedPerformance(
      aPerformance.playID,
      playFor(aPerformance),
      aPerformance.audience,
      amountFor(aPerformance),
      volumeCreditsFor(aPerformance))

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.volumeCredits
    result

  def totalAmount: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.amount
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

```
case class Performance(
  playID: String,
  audience: Int)
```

```
case class EnrichedPerformance(
  playID: String,
  play: Play,
  audience: Int,
  amount: Int,
  volumeCredits: Int)
```

```
case class StatementData(customer: String, performances: List[EnrichedPerformance])
```

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
    EnrichedPerformance(
      aPerformance.playID,
      playFor(aPerformance),
      aPerformance.audience,
      amountFor(aPerformance),
      volumeCreditsFor(aPerformance))

  val statementData =
    StatementData(invoice.customer,invoice.performances.map(enrichPerformance))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  def totalVolumeCredits: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.volumeCredits
    result

  def totalAmount: Int =
    var result = 0
    for (perf <- data.performances)
      result += perf.amount
    result

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(totalAmount/100)}\n"
  result += s"You earned $totalVolumeCredits credits\n"
  result
```

Finally, I move the two calculations of the totals.

Martin Fowler
@martinfowler

```
case class StatementData(
  customer: String,
  performances: List[EnrichedPerformance])
```

Splitting the Phases of Calculation and Formatting

```
def statement(invoice: Invoice, plays: Map[String, Play]): String =

  def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
    EnrichedPerformance(
      aPerformance.playID,
      playFor(aPerformance),
      aPerformance.audience,
      amountFor(aPerformance),
      volumeCreditsFor(aPerformance))

  def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
    var result = 0
    for (perf <- performances)
      result += perf.volumeCredits
    result

  def totalAmount(performances:List[EnrichedPerformance]): Int =
    var result = 0
    for (perf <- performances)
      result += perf.amount
    result

  val enrichedPerformances = invoice.performances.map(enrichPerformance)
  val statementData = StatementData(invoice.customer,
                                    enrichedPerformances,
                                    totalAmount(enrichedPerformances),
                                    totalVolumeCredits(enrichedPerformances))
  renderPlainText(statementData)

def renderPlainText(data: StatementData): String =

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(data.totalAmount/100)}\n"
  result += s"You earned ${data.totalVolumeCredits} credits\n"
  result
```

```
case class StatementData(
  customer: String,
  performances: List[EnrichedPerformance],
  totalAmount: Int,
  totalVolumeCredits: Int)
```

Splitting the Phases of Calculation and Formatting

```
def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
   var result = 0
   for (perf <- performances)
     result += perf.volumeCredits
   result

def totalAmount(performances:List[EnrichedPerformance]): Int =
  var result = 0
  for (perf <- performances)
    result += perf.amount
  result
```

Martin Fowler

@martinfowler

I can’t resist a couple quick shots of **Remove Loop with Pipeline**

```
def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
  performances.foldLeft(0)((total,perf) => total + perf.volumeCredits)

def totalAmount(performances:List[EnrichedPerformance]): Int =
  performances.foldLeft(0)((total,perf) => total + perf.amount)
```

Splitting the Phases of Calculation and Formatting

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  val enrichedPerformances = invoice.performances.map(enrichPerformance)
  val statementData = StatementData(invoice.customer,
                                    enrichedPerformances,
                                    totalAmount(enrichedPerformances),
                                    totalVolumeCredits(enrichedPerformances))
  renderPlainText(statementData)
```

Martin Fowler

@martinfowler

I now extract all the **first-phase code** into its own function.

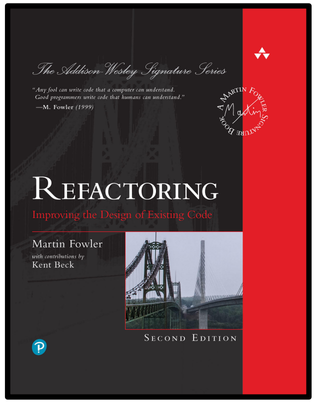
```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  renderPlainText(createStatementData(invoice, plays))

def createStatementData(invoice: Invoice, plays: Map[String, Play]): StatementData =
  val enrichedPerformances = invoice.performances.map(enrichPerformance)
  StatementData(invoice.customer,
                enrichedPerformances,
                totalAmount(enrichedPerformances),
                totalVolumeCredits(enrichedPerformances))
```

Note that on the previous slide, when we extracted **createStatementData**, all the functions nested inside **statement**, e.g. **totalAmount** and **totalVolumeCredits**, also moved along and are now nested in **createStatementData**.

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)



The Addison-Wesley Signature Series

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
—M. Fowler (1999)

A Martin Fowler Signature Book

Refactoring

Improving the Design of Existing Code

Martin Fowler
with contributions by
Kent Beck

Second Edition

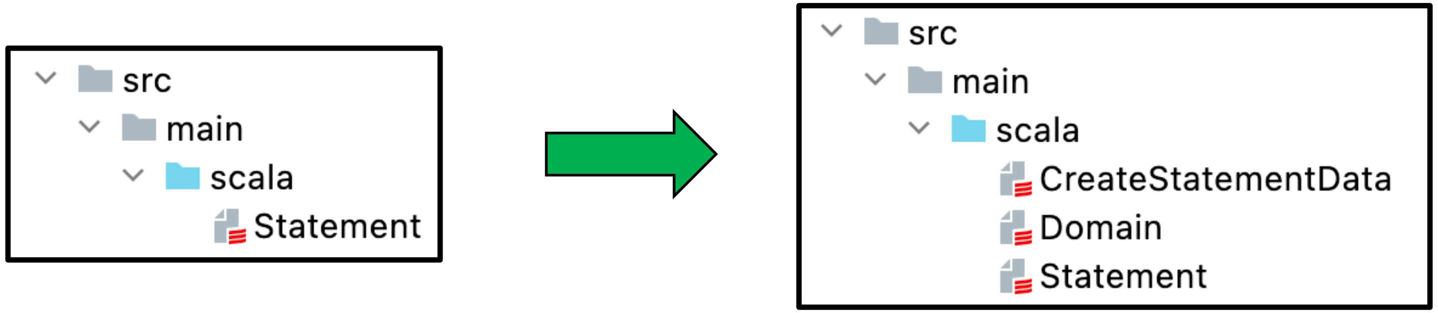
Status: Separated into Two Files (and Phases)

Martin Fowler

@martinfowler

Since **createStatementData** is now clearly separate, I move it into its own file.

See next slide – I also moved the **case classes** into their own file.



```
src
  main
    scala
      Statement
```

```
src
  main
    scala
      CreateStatementData
      Domain
      Statement
```

Status: Separated into Two Files (and Phases)

CreateStatementData.scala

```
def createStatementData(invoice: Invoice, plays: Map[String, Play]): StatementData =

  def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
    EnrichedPerformance(
      aPerformance.playID,
      playFor(aPerformance),
      aPerformance.audience,
      amountFor(aPerformance),
      volumeCreditsFor(aPerformance))

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def amountFor(aPerformance: Performance): Int =
    var result = 0
    playFor(aPerformance).`type` match
      case "tragedy" =>
        result = 40_000
        if aPerformance.audience > 30
        then result += 1_000 * (aPerformance.audience - 30)
      case "comedy" =>
        result = 30_000
        if aPerformance.audience > 20
        then result += 10_000 + 500 * (aPerformance.audience - 20)
        result += 300 * aPerformance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${playFor(aPerformance).`type`}")
    result

  def volumeCreditsFor(aPerformance: Performance) =
    var result = 0
    result += math.max(aPerformance.audience - 30, 0)
    if "comedy" == playFor(aPerformance).`type`
    then result += math.floor(aPerformance.audience / 5).toInt
    result

  def totalAmount(performances:List[EnrichedPerformance]): Int =
    performances.foldLeft(0)((total,perf) => total + perf.amount)

  def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
   performances.foldLeft(0)((total,perf) => total + perf.volumeCredits)

  val enrichedPerformances = invoice.performances.map(enrichPerformance)
  StatementData(invoice.customer,
                enrichedPerformances,
                totalAmount(enrichedPerformances),
                totalVolumeCredits(enrichedPerformances))
```

Statement.scala

```
import java.text.NumberFormat
import java.util.{Currency, Locale}
import scala.math

def statement(invoice: Invoice, plays: Map[String, Play]): String =
  renderPlainText(createStatementData(invoice, plays))

def renderPlainText(data: StatementData): String =

  def usd(aNumber: Int): String =
    val formatter = NumberFormat.getCurrencyInstance(Locale.US)
    formatter.setCurrency(Currency.getInstance(Locale.US))
    formatter.format(aNumber)

  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(data.totalAmount/100)}\n"
  result += s"You earned ${data.totalVolumeCredits} credits\n"
  result
```

Domain.scala

```
case class Performance(playID: String, audience: Int)

case class EnrichedPerformance(
  playID: String,
  play: Play,
  audience: Int,
  amount: Int,
  volumeCredits: Int)

case class Invoice(customer: String, performances: List[Performance])

case class Play(name: String, `type`: String)

case class StatementData(
  customer: String,
  performances: List[EnrichedPerformance],
  totalAmount: Int
  totalVolumeCredits: Int)
```

Status: Separated into Two Files (and Phases)

Martin Fowler

@martinfowler

It is now easy to write an **HTML** version of **statement** and **renderPlainText** (I moved **usd** to the top level so that renderHtml could use it).

```
def htmlStatement(invoice: Invoice, plays: Map[String, Play]): String =
  renderHtml(createStatementData(invoice,plays))

def renderHtml(data: StatementData): String =
  var result = s"<h1>Statement for ${data.customer}</h1>\n"
  result += "<table>\n"
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>\n"
  for (perf <- data.performances)
    result += s"<tr><td>${perf.play.name}</td><td>${perf.audience}</td>"
    result += s"<td>${usd(perf.amount/100)}</td></tr>\n"
  result += "</table>\n"
  result += s"<p>Amount owed is <em>${usd(data.totalAmount/100)}</em></p>\n"
  result += s"<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>\n"
  result
```

Let’s add an assertion test for **htmlStatement**.

@philip_schwarz

Status: Separated into Two Files (and Phases)

```
@main def main: Unit =
  assert(
    statement(invoices(0), plays)
    ==
    """|Statement for BigCo
       |  Hamlet: $650.00 (55 seats)
       |  As You Like It: $580.00 (35 seats)
       |  Othello: $500.00 (40 seats)
       |Amount owed is $1,730.00
       |You earned 47 credits
       |""".stripMargin
  )
  assert(
    htmlStatement(invoices(0), plays)
    ==
    """|<h1>Statement for BigCo</h1>
       |<table>
       |<tr><th>play</th><th>seats</th><th>cost</th></tr>
       |<tr><td>Hamlet</td><td>55</td><td>$650.00</td></tr>
       |<tr><td>As You Like It</td><td>35</td><td>$580.00</td></tr>
       |<tr><td>Othello</td><td>40</td><td>$500.00</td></tr>
       |</table>
       |<p>Amount owed is <em>$1,730.00</em></p>
       |<p>You earned <em>47</em> credits</p>
       |""".stripMargin
   )
```

```
val invoices: List[Invoice] = List(
  Invoice( customer = "BigCo",
           performances = List(Performance(playID = "hamlet",
                                           audience = 55),
                               Performance(playID = "as-like",
                                           audience = 35),
                               Performance(playID = "othello",
                                           audience = 40)))
)

val plays = Map (
  "hamlet"  -> Play(name = "Hamlet", `type` = "tragedy"),
  "as-like" -> Play(name = "As You Like It", `type` = "comedy"),
  "othello" -> Play(name = "Othello", `type` = "tragedy")
)
```

Martin Fowler
@martinfowler

There are more things I could do to simplify the printing logic, but this will do for the moment.

I always have to strike a balance between all the refactorings I could do and adding new features.

At the moment, most people under-prioritize refactoring—but there still is a **balance**.

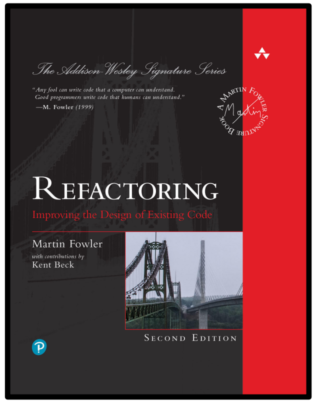
My rule is a variation on the camping rule:

Always leave the code base healthier than when you found it.

It will never be perfect, but it should be better.

Martin Fowler
@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Martin Fowler
@martinfowler

Now I'll turn my attention to the next feature change: supporting more categories of plays, each with its own charging and volume credits calculations. At the moment, to make changes here I have to go into the calculation functions and edit the conditions in there.

The amountFor function highlights the central role the type of play has in the choice of calculations—but conditional logic like this tends to decay as further modifications are made unless it's reinforced by more structural elements of the programming language.

There are various ways to introduce structure to make this explicit, but in this case a natural approach is type polymorphism—a prominent feature of classical object-orientation. Classical OO has long been a controversial feature in the JavaScript world, but the ECMAScript 2015 version provides a sound syntax and structure for it. So it makes sense to use it in a right situation—like this one.

My overall plan is to set up an inheritance hierarchy with comedy and tragedy subclasses that contain the calculation logic for those cases. Callers call a polymorphic amount function that the language will dispatch to the different calculations for the comedies and tragedies. I'll make a similar structure for the volume credits calculation. To do this, I utilize a couple of refactorings.

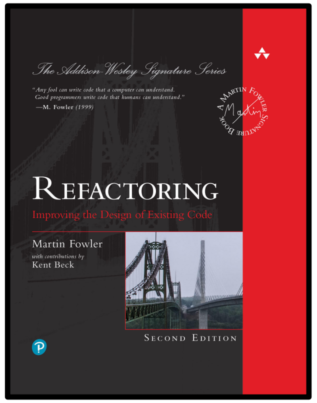
The core refactoring is Replace Conditional with Polymorphism, which changes a hunk of conditional code with polymorphism. But before I can do Replace Conditional with Polymorphism, I need to create an inheritance structure of some kind. I need to create a class to host the amount and volume credit functions.

```
def amountFor(aPerformance: Performance): Int =
   var result = 0
   playFor(aPerformance).`type` match
     case "tragedy" =>
       result = 40_000
       if aPerformance.audience > 30
       then result += 1_000 * (aPerformance.audience - 30)
     case "comedy" =>
       result = 30_000
       if aPerformance.audience > 20
       then result += 10_000 + 500 * (aPerformance.audience - 20)
       result += 300 * aPerformance.audience
     case other =>
       throw IllegalArgumentException(s"unknown type ${playFor(aPerformance).`type`}")
   result
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator


The Addison Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Creating a Performance Calculator

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  EnrichedPerformance(
    aPerformance.playID,
    playFor(aPerformance),
    aPerformance.audience,
    amountFor(aPerformance),
    volumeCreditsFor(aPerformance))
```

Martin Fowler
@martinfowler

The **enrichPerformance** function is the key, since it populates the intermediate data structure with the data for each **performance**.

Currently, it calls the conditional functions for **amount** and **volume credits**. What I need it to do is call those functions on a **host class**.

Since that class **hosts** functions for calculating data about **performances**, I'll call it a **performance calculator**.

```
case class PerformanceCalculator(performance: Performance)
```

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance)
  EnrichedPerformance(
    aPerformance.playID,
    playFor(aPerformance),
    aPerformance.audience,
    amountFor(aPerformance),
    volumeCreditsFor(aPerformance))
```

Creating a Performance Calculator

```
case class PerformanceCalculator(performance: Performance)
```

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance)
  EnrichedPerformance(
    aPerformance.playID,
    playFor(aPerformance),
    aPerformance.audience,
    amountFor(aPerformance),
    volumeCreditsFor(aPerformance))
```

Martin Fowler
@martinfowler

So far, this new object isn't doing anything. I want to move behavior into it—and I'd like to start with the simplest thing to move, which is the play record.

Strictly, I don't need to do this, as it's not varying polymorphically, but this way I'll keep all the data transforms in one place, and that consistency will make the code clearer.

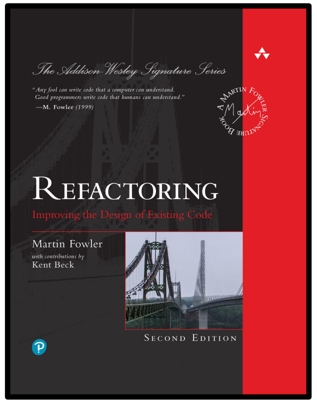
```
case class PerformanceCalculator(performance: Performance, play: Play)
```

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance,playFor(aPerformance))
  EnrichedPerformance(
    aPerformance.playID,
    calculator.play,
    aPerformance.audience,
    amountFor(aPerformance),
    volumeCreditsFor(aPerformance))
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator
    - Moving Functions into the Calculator



The Addison-Wesley Signature Series

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)

A Martin Fowler Signature Book

REFACTORING

Improving the Design of Existing Code

Martin Fowler
with contributions by
Kent Beck

SECOND EDITION

Moving Functions into the Calculator

```scala
case class PerformanceCalculator(performance: Performance, play: Play)
```

```scala
def amountFor(aPerformance: Performance): Int =
  var result = 0
  playFor(aPerformance).`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw IllegalArgumentException(s"unknown type ${playFor(aPerformance).`type`}")
  result
```

The next bit of logic I move is rather more substantial for calculating the amount for a performance…

The first part of this refactoring is to copy the logic over to its new context—the calculator class.

Then, I adjust the code to fit into its new home, changing aPerformance to performance and playFor(aPerformance) to play.

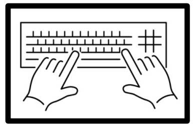
Martin Fowler
@martinfowler

Move Function amountFor

```scala
case class PerformanceCalculator(performance: Performance, play: Play):
  def amount: Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if performance.audience > 30
        then result += 1_000 * (performance.audience - 30)
      case "comedy" =>
        result = 30_000
        if performance.audience > 20
        then result += 10_000 + 500 * (performance.audience - 20)
        result += 300 * performance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result
```

Moving Functions into the Calculator

```scala
def amountFor(aPerformance: Performance): Int =
  var result = 0
  playFor(aPerformance).`type` match
    case "tragedy" =>
      result = 40_000
      if aPerformance.audience > 30
      then result += 1_000 * (aPerformance.audience - 30)
    case "comedy" =>
      result = 30_000
      if aPerformance.audience > 20
      then result += 10_000 + 500 * (aPerformance.audience - 20)
      result += 300 * aPerformance.audience
    case other =>
      throw new IllegalArgumentException(s"unknown type ${playFor(aPerformance).`type`}")
  result
```



Move Function amountFor (continued)

Martin Fowler

@martinfowler

Once the new function fits its home, I take the original function and turn it into a **delegating function** so it calls the new function.

```scala
def amountFor(aPerformance: Performance): Int =
  PerformanceCalculator(aPerformance,playFor(aPerformance)).amount
```

Moving Functions into the Calculator

```
def amountFor(aPerformance: Performance): Int =
  PerformanceCalculator(aPerformance,playFor(aPerformance)).amount
```

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance,playFor(aPerformance))
  EnrichedPerformance(
    aPerformance.playID,
    calculator.play,
    aPerformance.audience,
    amountFor(aPerformance),
    volumeCreditsFor(aPerformance))
```

Martin Fowler
@martinfowler

With that done, I use **Inline Function** to call the new **amount** function directly.

Yes, we are not just inlining **amountFor**, we are then taking into consideration the fact that the body of **amountFor** that we have just inlined is equivalent to the simpler expression **calculator.amount**.

Inline Function amountFor

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance,playFor(aPerformance))
  EnrichedPerformance(
    aPerformance.playID,
    calculator.play,
    aPerformance.audience,
    calculator.amount,
    volumeCreditsFor(aPerformance))
```

Moving Functions into the Calculator

```
def volumeCreditsFor(aPerformance: Performance) =
  var result = 0
  result += math.max(aPerformance.audience - 30, 0)
  if "comedy" == playFor(aPerformance).`type`
  then result += math.floor(aPerformance.audience / 5).toInt
  result
```

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance,playFor(aPerformance))
  EnrichedPerformance(
    aPerformance.playID,
    calculator.play,
    aPerformance.audience,
    calculator.amount,
    volumeCreditsFor(aPerformance))
```

Martin Fowler
@martinfowler

I repeat the same process to move the **volume credits calculation**.

Move Function volumeCreditsFor

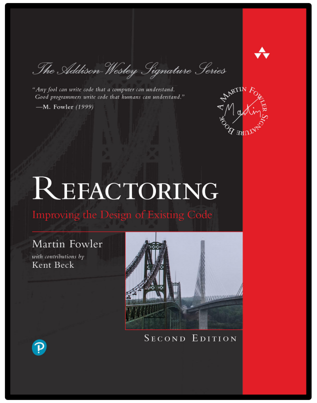
```
case class PerformanceCalculator(performance: Performance, play: Play):
  def amount: Int =
    …
  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result
```

```
def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
  val calculator = PerformanceCalculator(aPerformance,playFor(aPerformance))
  EnrichedPerformance(
    aPerformance.playID,
    calculator.play,
    aPerformance.audience,
    calculator.amount,
    calculator.volumeCredits)
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator
    - Moving Functions into the Calculator
    - Making the Performance Calculator Polymorphic


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Making the Performance Calculator Polymorphic

```scala
case class PerformanceCalculator(performance: Performance, play: Play):

  def amount: Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if performance.audience > 30
        then result += 1_000 * (performance.audience - 30)
      case "comedy" =>
        result = 30_000
        if performance.audience > 20
        then result += 10_000 + 500 * (performance.audience - 20)
        result += 300 * performance.audience
      case other =>
        throw new IllegalArgumentException(
          s"unknown type ${play.`type`}")
    result

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result
```

Now that I have the logic in a class, it's time to apply the polymorphism. The first step is to use Replace Type Code with Subclasses to introduce subclasses instead of the type code.

Martin Fowler

@martinfowler

```scala
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if performance.audience > 30
        then result += 1_000 * (performance.audience - 30)
      case "comedy" =>
        result = 30_000
        if performance.audience > 20
        then result += 10_000 + 500 * (performance.audience - 20)
        result += 300 * performance.audience
      case other =>
        throw new IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result

case class TragedyCalculator(performance: Performance, play: Play) extends PerformanceCalculator

case class ComedyCalculator(performance: Performance, play: Play) extends PerformanceCalculator

object PerformanceCalculator:
  def apply(aPerformance: Performance, aPlay: Play): PerformanceCalculator =
    aPlay.`type` match
      case "tragedy" => TragedyCalculator(aPerformance,aPlay)
      case "comedy" => ComedyCalculator(aPerformance,aPlay)
      case other => throw new IllegalArgumentException(s"unknown type ${aPlay.`type`}")
```

In Scala, we decided to map the superclass to an interface (trait), and the subclasses to implementations of the interface (trait).

Making the Performance Calculator Polymorphic

```
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int =
    var result = 0
    play.`type` match
      case "tragedy" =>
        result = 40_000
        if performance.audience > 30
        then result += 1_000 * (performance.audience - 30)
      case "comedy" =>
        result = 30_000
        if performance.audience > 20
        then result += 10_000 + 500 * (performance.audience - 20)
        result += 300 * performance.audience
      case other =>
        throw new IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
```

This sets up the **structure** for the **polymorphism**, so I can now move on to **Replace Conditional with Polymorphism**.

Martin Fowler

```
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int =
    var result = 0
    play.`type` match
      case "tragedy" => throw IllegalArgumentException(s”bad thing")
      case "comedy" =>
        result = 30_000
        if performance.audience > 20
        then result += 10_000 + 500 * (performance.audience - 20)
        result += 300 * performance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  override def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
```

Making the Performance Calculator Polymorphic

```
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int =
    var result = 0
    play.`type` match
      case "tragedy" => throw IllegalArgumentException(s”bad thing")
      case "comedy" =>
        result = 30_000
        if performance.audience > 20
        then result += 10_000 + 500 * (performance.audience - 20)
        result += 300 * performance.audience
      case other =>
        throw IllegalArgumentException(s"unknown type ${play.`type`}")
    result

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  override def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
```

```
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
  def amount: Int =
    var result = 30_000
    if performance.audience > 20
    then result += 10_000 + 500 * (performance.audience - 20)
    result += 300 * performance.audience
    result
```

Now I move the **comedy** **case** down too.

Martin Fowler

@martinfowler

Making the Performance Calculator Polymorphic

```
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int

  def volumeCredits: Int =
    var result = 0
    result += math.max(performance.audience - 30, 0)
    if "comedy" == play.`type`
    then result += math.floor(performance.audience / 5).toInt
    result

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  override def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
  override def amount: Int =
    var result = 30_000
    if performance.audience > 20
    then result += 10_000 + 500 * (performance.audience - 20)
    result += 300 * performance.audience
    result
```

The next **conditional** to replace is the **volumeCredits calculation**.

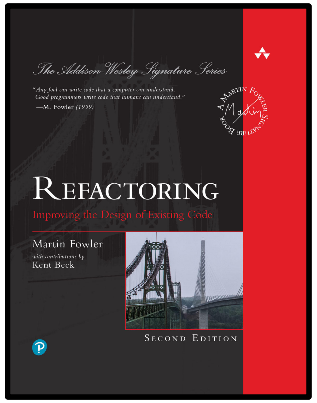
Martin Fowler

@martinfowler

```
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int

  def volumeCredits: Int = math.max(performance.audience - 30, 0)

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
  def amount: Int =
    var result = 30_000
    if performance.audience > 20
    then result += 10_000 + 500 * (performance.audience - 20)
    result += 300 * performance.audience
    result
  override def volumeCredits: Int =
    super.volumeCredits + math.floor(performance.audience / 5).toInt
```

Martin Fowler
@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator
    - Moving Functions into the Calculator
    - Making the Performance Calculator Polymorphic
  - Status: Creating the Data with the Polymorphic Calculator



Martin Fowler
@martinfowler

Time to reflect on what introducing the polymorphic calculator did to the code.

Again, the code has increased in size as I've introduced structure.

The benefit here is that the calculations for each kind of play are grouped together.

If most of the changes will be to this code, it will be helpful to have it clearly separated like this.

Adding a new kind of play requires writing a new subclass and adding it to the creation function.

The example gives some insight as to when using subclasses like this is useful.

See next slide for the initial code.

See the three subsequent slides for the refactored code.

Initial Program

```scala
def statement(invoice: Invoice, plays: Map[String, Play]): String =
  var totalAmount = 0
  var volumeCredits = 0
  var result = s"Statement for ${invoice.customer}\n"
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))

  for (perf <- invoice.performances)
    val play = plays(perf.playID)
    var thisAmount = 0

    play.`type` match
      case "tragedy" =>
        thisAmount = 40_000
        if perf.audience > 30
        then thisAmount += 1_000 * (perf.audience - 30)
      case "comedy" =>
        thisAmount = 30_000
        if perf.audience > 20
        then thisAmount += 10_000 + 500 * (perf.audience - 20)
        thisAmount += 300 * perf.audience
      case other =>
        throw IllegalArgumentException(s"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`
    then volumeCredits += math.floor(perf.audience / 5).toInt

    // print line for this order
    result += s"  ${play.name}: ${formatter.format(thisAmount/100)} (${perf.audience} seats)\n"
    totalAmount += thisAmount
  end for

  result += s"Amount owed is ${formatter.format(totalAmount/100)}\n"
  result += s"You earned $volumeCredits credits\n"
  result
```

```scala
case class Play(name: String, `type`: String)
```

```scala
case class Performance(playID: String, audience: Int)
```

```scala
case class Invoice(
  customer: String,
  performances: List[Performance]
)
```

Statement.scala

Refactored Program

```scala
import java.text.NumberFormat
import java.util.{Currency, Locale}
import scala.math

def statement(invoice: Invoice, plays: Map[String, Play]): String =
  renderPlainText(createStatementData(invoice, plays))

def htmlStatement(invoice: Invoice, plays: Map[String, Play]): String =
  renderHtml(createStatementData(invoice,plays))

def renderPlainText(data: StatementData): String =
  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(data.totalAmount/100)}\n"
  result += s"You earned ${data.totalVolumeCredits} credits\n"
  result

def renderHtml(data: StatementData): String =
  var result = s"<h1>Statement for ${data.customer}</h1>\n"
  result += "<table>\n"
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>\n"
  for (perf <- data.performances)
    result += s"<tr><td>${perf.play.name}</td><td>${perf.audience}</td>"
    result += s"<td>${usd(perf.amount/100)}</td></tr>\n"
  result += "</table>\n"
  result += s"<p>Amount owed is <em>${usd(data.totalAmount/100)}</em></p>\n"
  result += s"<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>\n"
  result

def usd(aNumber: Int): String =
  val formatter = NumberFormat.getCurrencyInstance(Locale.US)
  formatter.setCurrency(Currency.getInstance(Locale.US))
  formatter.format(aNumber)
```

Refactored Program

CreateStatementData.scala

```
def createStatementData(invoice: Invoice, plays: Map[String, Play]): StatementData =

  def enrichPerformance(aPerformance: Performance): EnrichedPerformance =
   val calculator = PerformanceCalculator(aPerformance,playFor(aPerformance))
   EnrichedPerformance(
     aPerformance.playID,
     calculator.play,
     aPerformance.audience,
     calculator.amount,
     calculator.volumeCredits)

  def playFor(aPerformance: Performance): Play =
    plays(aPerformance.playID)

  def totalAmount(performances:List[EnrichedPerformance]): Int =
    performances.foldLeft(0)((total,perf) => total + perf.amount)

  def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
   performances.foldLeft(0)((total,perf) => total + perf.volumeCredits)

  val enrichedPerformances = invoice.performances.map(enrichPerformance)
  StatementData(invoice.customer,
                enrichedPerformances,
                totalAmount(enrichedPerformances),
                totalVolumeCredits(enrichedPerformances))
```

Domain.scala

```scala
case class Performance(playID: String, audience: Int)

case class EnrichedPerformance(
  playID: String,
  play: Play,
  audience: Int,
  amount: Int,
  volumeCredits: Int)

case class Invoice(customer: String, performances: List[Performance])

case class Play(name: String, `type`: String)

case class StatementData(
  customer: String,
  performances: List[EnrichedPerformance],
  totalAmount: Int
  totalVolumeCredits)
```

```scala
sealed trait PerformanceCalculator :
  def performance: Performance
  def play: Play
  def amount: Int
  def volumeCredits: Int = math.max(performance.audience - 30, 0)

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
  def amount: Int =
    var result = 30_000
    if performance.audience > 20
    then result += 10_000 + 500 * (performance.audience - 20)
    result += 300 * performance.audience
    result
  override def volumeCredits: Int =
    super.volumeCredits + math.floor(performance.audience / 5).toInt
```

To conclude this slide deck, let's make three more small improvements to the **Scala** code.

First, let's get rid of the remaining **mutability** in the **calculation logic**.

```scala
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int

  def volumeCredits: Int = math.max(performance.audience - 30, 0)

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  override def amount: Int =
    var result = 40_000
    if performance.audience > 30
    then result += 1_000 * (performance.audience - 30)
    result

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator
  override def amount: Int =
    var result = 30_000
    if performance.audience > 20
    then result += 10_000 + 500 * (performance.audience - 20)
    result += 300 * performance.audience
    result
  override def volumeCredits: Int =
    super.volumeCredits + math.floor(performance.audience / 5).toInt
```

```scala
sealed trait PerformanceCalculator :

  def performance: Performance

  def play: Play

  def amount: Int

  def volumeCredits: Int = math.max(performance.audience - 30, 0)

case class TragedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  def amount: Int =
    val basicAmount = 40_000
    val largeAudiencePremiumAmount =
      if performance.audience <= 30 then 0
      else 1_000 * (performance.audience - 30)
    basicAmount + largeAudiencePremiumAmount

case class ComedyCalculator(performance: Performance, play: Play)
extends PerformanceCalculator:
  def amount: Int =
    val basicAmount = 30_000
    val largeAudiencePremiumAmount =
      if performance.audience <= 20 then 0
      else 10_000 + 500 * (performance.audience - 20)
    val audienceSizeAmount = 300 * performance.audience
    basicAmount + largeAudiencePremiumAmount + audienceSizeAmount
  override def volumeCredits: Int =
    super.volumeCredits + math.floor(performance.audience / 5).toInt
```

Next, let's get rid of the **mutability** in the **rendering logic**.

```
def renderPlainText(data: StatementData): String =
  var result = s"Statement for ${data.customer}\n"
  for (perf <- data.performances)
    result +=
      s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  result += s"Amount owed is ${usd(data.totalAmount/100)}\n"
  result += s"You earned ${data.totalVolumeCredits} credits\n"
  result
```

```
def renderPlainText(data: StatementData): String =
  s"Statement for ${data.customer}\n" + (
    for
      perf <- data.performances
    yield s"  ${perf.play.name}: ${usd(perf.amount/100)} (${perf.audience} seats)\n"
  ).mkString +
  s"""|Amount owed is ${usd(data.totalAmount/100)}
      |You earned ${data.totalVolumeCredits} credits
      |""".stripMargin
```

```
def renderHtml(data: StatementData): String =
  var result = s"<h1>Statement for ${data.customer}</h1>\n"
  result += "<table>\n"
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>\n"
  for (perf <- data.performances)
    result += s"<tr><td>${perf.play.name}</td><td>${perf.audience}</td>"
    result += s"<td>${usd(perf.amount/100)}</td></tr>\n"
  result += "</table>\n"
  result += s"<p>Amount owed is <em>${usd(data.totalAmount/100)}</em></p>\n"
  result += s"<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>\n"
  result
```

```
def renderHtml(data: StatementData): String =
  s"""|<h1>Statement for ${data.customer}</h1>
      |<table>
      |<tr><th>play</th><th>seats</th><th>cost</th></tr>
      |""".stripMargin + (
    for
      perf <- data.performances
    yield  s"<tr><td>${perf.play.name}</td><td>${perf.audience}</td>" +
           s"<td>${usd(perf.amount/100)}</td></tr>\n"
  ).mkString +
  s"""|</table>
      |<p>Amount owed is <em>${usd(data.totalAmount/100)}</em></p>
      |<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>
      |""".stripMargin
```

@philip_schwarz

And finally, let’s make a small change to increase the **readability** of the **totalling functions** for **amount** and **volume credits**.

```scala
def totalAmount(performances:List[EnrichedPerformance]): Int =
  performances.foldLeft(0)((total,perf) => total + perf.amount)

def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
  performances.foldLeft(0)((total,perf) => total + perf.volumeCredits)
```

```scala
def totalAmount(performances:List[EnrichedPerformance]): Int =
  performances.map(_.amount).sum

def totalVolumeCredits(performances:List[EnrichedPerformance]): Int =
  performances.map(_.volumeCredits).sum
```

That’s all.

I hope you found it useful.