# 1. 첫 번째 예시

**첫 번째 예시는 다음과 같다.**

- 다양한 연극을 외주로 받아서 공연하는 극단이 있다.
- 공연 요청이 들어오면 연극의 장르와 관객 규모를 기초로 비용을 책정한다.
- 현재 이 극단은 두 가지 장르, 비극(tragedy)과 희극(comedy)만 공연한다.
- 그리고 공연료와 별개로 포인트(volume credit)를 지급해서 다음번 의뢰 시 공연료를 할인 받을 수도 있다. 일종의 충성도 프로그램인 셈이다.

▼ 첫 번째 예시 코드 (ver 1.1)

```
const plays = require('./data/plays.json');
const invoice = require('./data/invoices.json');

// 공연료 청구서를 출력하는 코드
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `청구 내역 (고객명: ${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": // 비극
        thisAmount = 40000;
        if (perf.audience > 30) {
          thisAmount += 1000 * (perf.audience - 30);
        }
        break;
      case "comedy": // 희극
        thisAmount = 30000;
        if (perf.audience > 20) {
          thisAmount += 10000 + 500 * (perf.audience - 20);
        }
        thisAmount += 300 * perf.audience;
```

```
        break;
      default:
        throw new Error(`알 수 없는 장르: ${play.type}`);
    }
    // 포인트 적립
    volumeCredits += Math.max(perf.audience - 30, 0);
    // 희극 관객 5명마다 추가 포인트를 제공
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // 청구 내역 출력
    result += ` ${play.name}: ${format(thisAmount / 100)} (${
      perf.audience
    }석)\n`;
    totalAmount += thisAmount;
  }
    result += `총액: ${format(totalAmount/100)}\n`;
    result += `적립 포인트: ${volumeCredits}점\n`;
    return result;
}

console.log(statement(invoice, plays));
```

## 1.2 예시 프로그램을 본 소감

- 프로그램이 짧아 특별히 이해해야 할 구조가 없지만, 이런 코드가 수백 줄짜리 프로그램의 일부라면 간단한 인라인 함수 하나라도 이해하기가 쉽지 않다.
- 프로그램이 잘 작동하는 상황에서 미적 기준만으로 `나쁜 코드` 라고 판단할 수 있을까? 컴파일러는 코드가 깔끔하든 지저분하든 개의치 않으니 말이다.
- 하지만 **코드를 수정하려면 사람이 개입되고, 사람은 코드의 미적 상태에 민감하다.**
- 설계가 나쁜 시스템은 수정하기 어렵다. 무엇을 수정해야할지 쉽게 찾을 수 없다면 실수를 야기하게 되고 그 결과 버그가 생길 가능성이 높아진다.

> 💡 **프로그램이 새로운 기능을 추가하기에 편한 구조가 아니라면, 먼저 기능을 추가하기 쉬운 형태로 리팩터링하고 나서 원하는 기능을 추가한다.**

### 수정 및 기능 추가 사항

- 청구 내역을 HTML로 출력하는 기능
- 연극 장르와 공연료 정책 변화에 유연한 구조

## 1.3 리팩터링의 첫 단계

- **리팩터링의 첫 단계는 항상 `테스트 코드 작성` 이다.**
  - 리팩터링할 코드 영역을 꼼꼼하게 검사해줄 테스트 코드들부터 마련해야 한다.
  - 프로그램이 클수록 수정 과정에서 예상치 못한 문제가 발생할 가능성이 크기 때문에 테스트 코드를 마련하여 버그 발생 여부를 지속적으로 체크해야한다.
- **`statement()` 함수의 테스트 구성은?**
  - 다양한 장르의 공연들로 구성된 공연료 청구서 몇 개를 미리 작성하여 문자열 형태로 준비
  - `statement()` 가 반환한 문자열과 준비해둔 정답 문자열을 비교
  - 테스트 프레임워크를 이용하여 모든 테스트를 단축키 하나로 실행할 수 있도록 설정
- **여기서 중요한 부분은 테스트 결과를 보고하는 방식!**
  - 성공/실패 를 스스로 판단하는 자가진단 테스트로 만든다.

> 💡 **리팩터링하기 전에 제대로 된 테스트부터 마련한다. 테스트는 반드시 자가진단하도록 만든다.**

- 테스트를 작성하는 데 시간이 좀 걸리지만, 신경 써서 만들어두면 디버깅 시간이 줄어서 전체 작업 시간은 오히려 단축된다.

## 1.4 statement() 함수 쪼개기

`statement()` 처럼 긴 함수를 리팩터링할 때는 먼저 `전체 동작은 각각의 부분으로 나눌 수 있는 지점을 찾는다.` 그러면 중간 즈음의 `switch`문이 가장 먼저 눈에 띌 것이다.

▼ **[1] switch문 추출하기**

- `switch`문을 살펴보면 한 번의 공연에 대한 요금을 계산하고 있다.
- 이처럼 코드를 분석해서 얻은 정보들은 재빨리 코드에 반영해야한다.
  (이후, 다시 분석하지 않아도 코드 스스로가 자신이 하는 일이 무엇인지 얘기해준다)

- 추출한 함수에는 그 코드가 하는 일을 설명하는 이름을 지어준다.
  ⇒ `amountFor(aPerformance)`

### 코드 조각을 함수로 추출할 때 실수를 최소화하기 위한 절차

**- 유효범위를 벗어나는 변수, 즉 새 함수에서는 곧바로 사용할 수 없는 변수 체크**
⇒ 이번 예에서는 `perf, play, thisAmount` 가 여기 속한다.
⇒ `perf, play` 는 추출한 새 함수에서도 필요하지만 값을 변경하지 않기 때문에 매개변수로 전달하면 된다.
⇒ `thisAmount` 는 함수 안에서 값이 바뀌는데, 이런 변수는 조심히 다뤄야 한다. 이번 예에서는 이런 변수가 하나뿐이므로 이 값을 반환하도록 작성한다.

**- 추출한 함수를 보다 명확하게 표현할 수 있는 방법이 없는지 검토**
⇒ 변수 이름을 좀 더 명확하게 바꿔보자. `thisAmount` → `result` 로 수정
⇒ 필자는 함수의 반환 값에 항상 `result` 라는 이름을 쓴다.
⇒ 첫 번째 인수인 `perf` → `aPerformance` 로 리팩터링
⇒ 자바스크립트와 같은 동적 타입 언어를 사용할 때는 타입이 드러나게 작성하면 도움이 된다. 지금처럼 매개변수의 역할이 뚜렷하지 않을 때는 부정 관사(a/an)을 붙인다.
⇒ 명확성을 높이기 위한 이름 바꾸기에는 조금도 망설이지 말아라.

**- 임시 변수를 질의 함수로 바꾸기**
⇒ 임시 변수들 때문에 로컬 범위에 존재하는 이름이 늘어나서 추출 작업이 복잡해지는 문제를 완화
⇒ `amountFor()` 의 매개변수 중 `play` 는 `개별공연(aPerformance)` 에서 얻기 때문에 애초에 매개 변수로 전달할 필요가 없다.
⇒ 대입문의 우변을 함수로 추출 → `playFor(aPerformance)`
⇒ 추출한 함수에 대해 최상위 함수 `statement()` 에서 `변수 인라인` 과정 진행
⇒ `amountFor()` 에 `함수 선언 바꾸기` 를 적용해서 `play` 매개변수 제거
⇒ 지역 변수를 제거해서 얻는 가장 큰 장점은 추출 작업이 훨씬 쉬워진다는 것! (유효범위를 신경 써야 할 대상이 줄어들기에. 필자는 추출 작업 전에는 거의 항상 지역 변수부터 제거한다)

**- 추출한 함수에 대한 변수 인라인**
⇒ `amountFor()` 는 임시 변수인 `thisAmount` 에 값을 설정할 때 사용되는데, 그 값이 다시 바뀌지 않으므로 `변수 인라인하기` 를 적용한다.

> 💡 **리팩터링은 프로그램 수정을 작은 단계로 나눠 진행한다.**
> **그러면 중간에 실수하더라도 버그를 쉽게 찾을 수 있다.**
>
> - 아무리 간단한 수정이라도 리팩터링 후에는 항상 테스트하는 습관을 들이자.
> - 조금씩 변경하고 매번 테스트하는 것은 리팩터링 절차의 핵심이다.

> 💡 **컴퓨터가 이해하는 코드는 바보도 작성할 수 있다. 사람이 이해하도록 작성하는 프로그래머가 진정한 실력자다.**

### ▼ [2] 적립 포인트 계산 코드 추출하기

- `volumeCredits`는 반복문을 돌 때마다 값을 누적해야 하기 때문에 까다롭다.
- 이 상황에서 최선의 방법은 추출한 함수에서 `volumeCredits`의 복제본을 초기화한 뒤 계산 결과를 반환토록 하는 것이다.
- **함수를 추출한 후, 새로 추출한 함수에서 쓰이는 변수들 이름을 적절히 바꾼다.**
  - 각 과정마다 `컴파일-테스트-커밋` 과정을 거친다.

```
// 적립 포인트 계산 코드 추출
function volumeCreditsFor(aPerformance) {
  let volumeCredits = 0;
  volumeCredits += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type)
    volumeCredits += Math.floor(aPerformance.audience / 5);
  return volumeCredits;
}
```

### ▼ [3] format 변수 제거하기

- 앞서 설명했듯이, 임시 변수는 나중에 문제를 일으킬 수 있다. 임시 변수는 자신이 속한 루틴에서만 의미가 있어서 루틴이 길고 복잡해지기 쉽다.
- 따라서 다음으로 할 리팩터링은 이런 변수들을 제거하는 것이다.
- 현재 `format`은 임시 변수에 함수를 대입한 형태인데, 이를 함수로 직접 선언해 바꿔보자.

- 추출한 함수의 이름 `format` 이 함수가 하는 일을 충분히 설명해주지 못한다. 이 함수의 핵심은 `화폐 단위 맞추기` 이므로 이러한 느낌을 살려 이름은 적절히 바꿔주는 `함수 선언 바꾸기` 를 적용해보자.
- 이름짓기는 중요하면서도 쉽지 않은 작업이다. 긴 함수를 작게 쪼개는 리팩터링은 이름을 잘 지어야만 효과가 있다. 단번에 좋은 이름을 짓기는 쉽지 않으므로, 처음에 떠오르는 최선의 이름을 사용하다가 나중에 더 좋은 이름이 떠오를 때 바꾸는 식이 좋다.
- 100으로 나누는 코드도 중복이 되므로 추출한 함수로 옮긴다.

```
// 화폐 단위를 맞추는 부분을 함수로 추출
function usd(aNumber) {
  return new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 2,
  }).format(aNumber/100);
}
```

### ▼ [4] volumeCredits 변수 제거하기

- `volumeCredits` 변수는 반복문을 한 바퀴 돌 때마다 값을 누적하기 때문에 리팩터링하기가 까다롭다.
- 따라서 먼저 `반복문 쪼개기` 로 `volumeCredits` 값이 누적되는 부분을 따로 빼낸다.
- 이어서 `문장 슬라이드하기` 를 적용해서 `volumeCredits` 변수를 선언하는 문장을 반복문 바로 앞으로 옮긴다.
- `volumeCredits` 값 갱신과 관련한 문장들을 한데 모아두면 `임수 변수를 질의 함수로 바꾸기` 가 수월해진다. 이번에도 역시 `volumeCredits` 값 계산 코드를 함수로 추출하는 작업 진행

```
function totalVolumeCredits() {
  let volumeCredits = 0;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  return volumeCredits
}

// 공연료 청구서를 출력하는 코드
function statement(invoice, plays) {
  let totalAmount = 0;
```

```
  let result = `청구 내역 (고객명: ${invoice.customer})\n`;

  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(aPerformance))} (${
      perf.audience
    }석)\n`;
    totalAmount += amountFor(aPerformance);
  }
  result += `총액: ${usd(totalAmount)}\n`;
  result += `적립 포인트: ${totalVolumeCredits()}점\n`;
  return result;
}
```

- 위와 같이 리팩터링을 진행하면, 반복문이 불필요하게 쪼개지기 때문에 성능이 느려지지 않을까 걱정할 수 있다. 이처럼 반복문이 중복되는 것을 꺼리는 이들이 많지만, 이 정도 중복은 성능에 미치는 영향이 미미할 때가 많다. 소프트웨어 성능은 대체로 코드의 몇몇 작은 부분에 의해 결정되므로 그 외의 부분은 수정한다고 해도 성능 차이를 체감할 수 없다.

> 💡 **리팩터링으로 인한 성능 문제 → '특별한 경우가 아니라면 일단 무시하라'**
>
> - 리팩터링이 성능에 상당한 영향을 준다고 할지라도 일단 리팩터링을 진행한다.
> - 잘 다듬어진 코드라야 성능 개선 작업도 훨씬 수월하기 때문이다.
> - 리팩터링 과정에서 성능이 크게 떨어졌다면 리팩터링 후 시간을 내어 성능을 개선
> - 리팩터링 덕분에 성능 개선을 더 효과적으로 수행할 수 있으며, 결과적으로 더 깔끔하면서 더 빠른 코드를 얻게 된다.

> 💡 **`volumeCredits` 변수를 제거하는 작업의 단계**
>
> 1. `반복문 쪼개기` 로 변수 값을 누적시키는 부분을 분리.
> 2. `문장 슬라이드하기` 로 변수 초기화 문장을 변수 값 누적 코드 바로 앞으로 옮김.
> 3. `함수 추출하기` 로 적립 포인트 계산 부분을 별도 함수로 추출.
> 4. `변수 인라인하기` 로 `volumeCredits` 변수를 제거.

- 커밋을 단계마다 수행하면, 중간에 테스트가 실패하고 원인을 바로 찾지 못할지라도 최근 커밋으로 돌아가서 다시 리팩터링을 시도한다.

▼ **[5] totalAmount 변수 제거하기**

- 위에서 변수를 제거했던 과정과 똑같은 절차로 진행
- 먼저 반복문을 쪼개고, 변수 초기화 문장을 옮긴 다음, 함수를 추출한다.
- 이후 `totalAmount` 변수를 인라인한 다음, 함수 이름을 더 의미 있게 고친다.

```
function totalAmount() {
  let result = 0;

  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(aPerformance))} (${
      perf.audience
    }석)\n`;
    result += amountFor(aPerformance);
  }
  return result;
}
```

# 1.5 중간 점검: 난무하는 중첩 함수

▼ 1차 리팩터링 진행 코드

```
const plays = require("./data/plays.json");
const invoice = require("./data/invoices.json");

// play 변수 제거하기
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}

// switch문 코드 조각 추출
function amountFor(aPerformance) {
  let result = 0;

  switch (playFor(aPerformance).type) {
    case "tragedy": // 비극
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
```

```
    case "comedy": // 희극
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`알 수 없는 장르: ${playFor(aPerformance).type}`);
  }
  return result;
}

// 적립 포인트 계산 코드 추출
function volumeCreditsFor(aPerformance) {
  let volumeCredits = 0;
  volumeCredits += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type)
    volumeCredits += Math.floor(aPerformance.audience / 5);
  return volumeCredits;
}

// 화폐 단위를 맞추는 부분을 함수로 추출
function usd(aNumber) {
  return new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 2,
  }).format(aNumber / 100);
}

// 포인트 계산하는 부분을 함수로 추출
function totalVolumeCredits() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}

// 총액 계산하는 부분을 함수로 추출
function totalAmount() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += amountFor(perf);
  }
  return result;
}

// 공연료 청구서를 출력하는 코드
function statement(invoice, plays) {
  let result = `청구 내역 (고객명: ${invoice.customer})\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${
```

```
      perf.audience
    }석)\n`;
  }
  result += `총액: ${usd(totalAmount())}\n`;
  result += `적립 포인트: ${totalVolumeCredits()}점\n`;
  return result;
}

console.log(statement(invoice, plays));
```

처음과 비교했을 때 코드 구조가 한결 나아졌다. 최상위의 `statement()` 함수는 이제 단 아홉 줄뿐이며, 출력할 문장을 생성하는 일만 한다. 계산 로직은 모두 여러 개의 보조 함수로 빼냈다. 결과적으로 각 계산 과정은 물론 전체 흐름을 이해하기가 훨씬 쉬워졌다.

## 1.6 계산 단계와 포맷팅 단계 분리하기