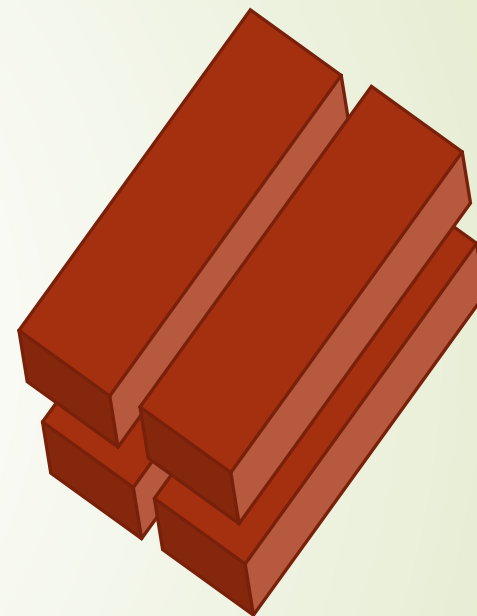
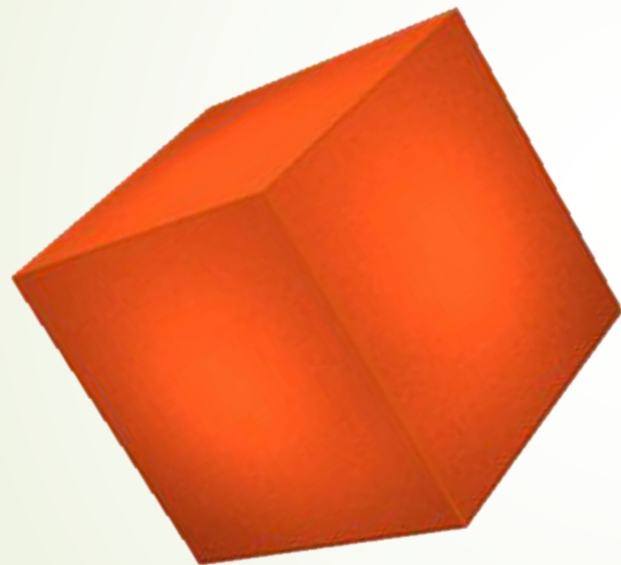




Функциональное программирование

Императивное программирование



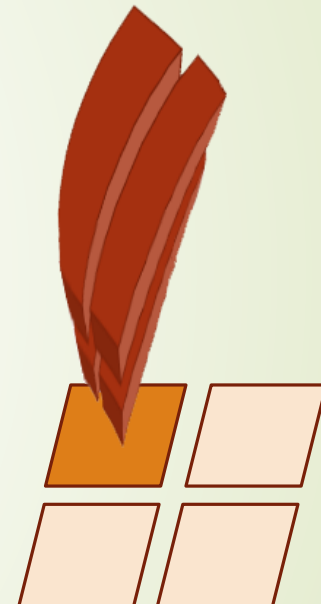
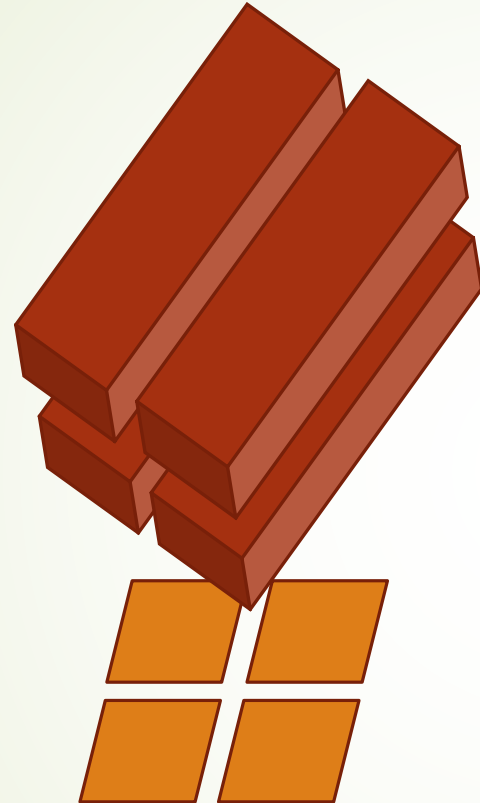
Разделяет проблему на отдельные части

Все было хорошо, пока в 2007 году не появились первые многоядерные процессоры



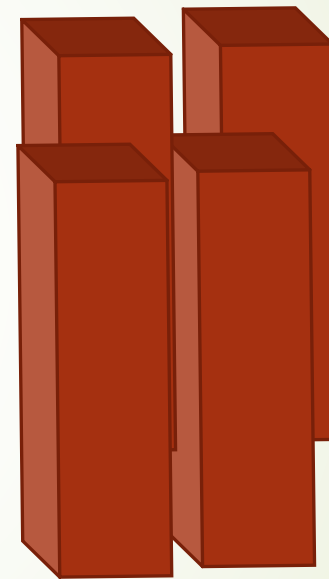
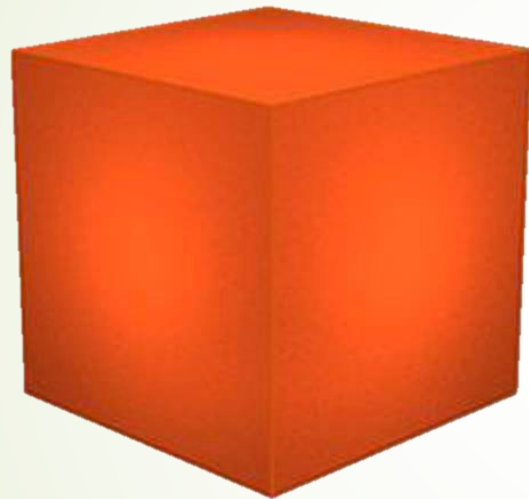
Нелегко стыковать, значит...

...надо втиснуть в одно ядро



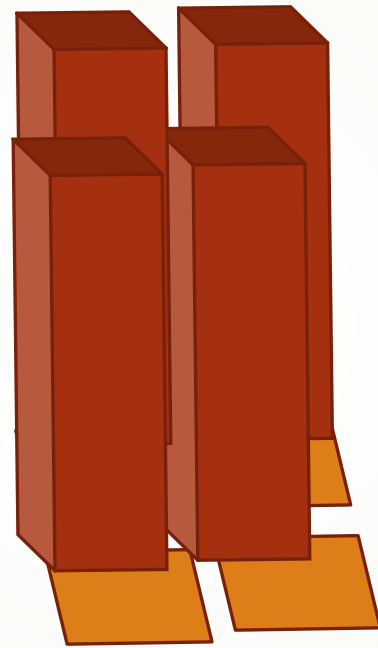
Расчлененная на отдельные части императивная программа не вписывается в многоядерный чип так, как надо.

Функциональное программирование



Разделяет проблему на отдельные части иначе

Каждая часть отлично стыкуется с ядром



Отдельные части функциональной программы
аккуратно вписываются в многоядерный чип

Решение старым способом

Предположим мы поддерживаем сайт, торгующий книгами, DVD и другими аналогичными товарами.

Каждая продажа характеризуется двумя свойствами:
item(товар) и price(цена)

Класс Sale

```
public class Sale{
    String item;
    double price;

    public Sale(String item, double price) {
        this.item = item;
        this.price = price;
    }
}
```

Решение старым способом

Программа вычисления общей суммы продаж DVD.

Использование класса Sale

```
public class TallySales{
    Public static void main(String[] args) {
        ArrayList<Sale> sales = new ArrayList<Sale>();
        NumberFormat currency = NumberFormat.getCurrencyInstance();

        fillTheList(sales);
        double total = 0;
        for (Sale sale: sales) {
            if (sale.item.equals("DVD")){
                total += sale.price;
            }
        }
        System.out.println(currency.format(total));
    }

    Static void fillTheList(ArrayList<Sale> sales) {
        sales.add(new Sale("DVD",15.00));
        sales.add(new Sale("Книга",12.00));
        sales.add(new Sale("DVD",21.00));
        sales.add(new Sale("CD",5.25));
    }
}
```


Другие возможные примеры

- Имеется список сотрудников. Необходимо отобрать сотрудников, оценка эффективности работы которых не ниже 3. Каждому такому сотруднику программа выписывает премию в размере 100 долларов и вычисляет суммарный размер премий по предприятию.
- Имеется список клиентов. Каждому клиенту, проявившему интерес к покупке смартфона, необходимо направить электронное письмо с информацией о скидках, действующих в текущем месяце.
- Имеется список открытых планет. Для каждой планеты класса «М» необходимо оценить вероятность существования на ней разумной жизни. После этого необходимо вычислить среднее значение всех вероятностных оценок.

Лямбда-выражения



В 1930-х годах математик Алонзо Черч использовал греческую букву «лямбда» (λ) для представления некоторой математической конструкции, которая создается на лету.

Помимо прочего, его система лямбда-исчислений легла в основу функциональных языков программирования, в частности семейства Лисп.

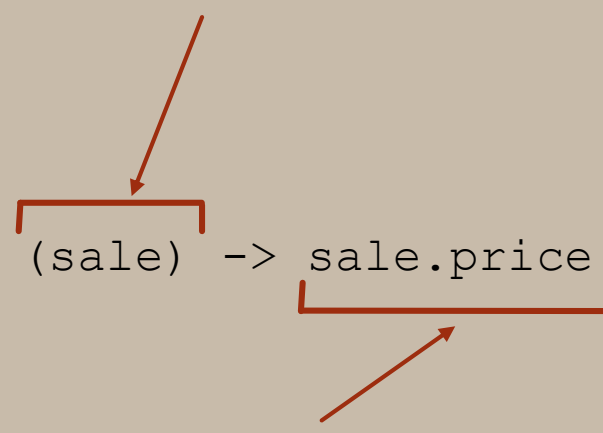
В наши дни в Java 8 термин *лямбда-выражение* представляет собой небольшой фрагмент кода, который служит одновременно и объявлением и вызовом метода, создаваемого на лету.

A stylized logo for the LISP programming language. The word "LISP" is written in a bold, sans-serif font. The letter "I" is replaced by a large, dark green, stylized shape that resembles a drop or a teardrop, with a white outline. The letters "L", "S", and "P" are in a dark green color, matching the "I" shape.

Простейшее лямбда-выражение

(sale) -> sale.price

1. Выбрать очередной объект из входного потока. Временно присвоить переменной *sale* ссылку на этот объект.



(sale) -> sale.price

2. Получить значение поля *price* данного объекта *sale*.

Простейшее лямбда-выражение

```
(sale) -> sale.price
```

Лямбда-выражение – сжатая запись объявления метода и его вызова без присвоения ему имени.

```
double getPrice(Sale sale) {  
    return sale.price();  
}
```


```
getPrice(sale);
```

Еще пример лямбда-выражения

```
(sale) -> sale.item.equals("DVD")
```

1. Выбрать очередной объект из входного потока. Временно присвоить переменной *sale* ссылку на этот объект.

```
(sale) -> sale.item.equals("DVD")
```



2. Получить значение поля *item* данного объекта *sale*.

3. Проверить, чему равен результат сравнения значения поля *item* объекта *sale* и строки "DVD": *true* или *false*.

Еще пример лямбда-выражения

```
(sale) -> sale.item.equals("DVD")
```

Лямбда-выражение – сжатая запись объявления метода и его вызова без присвоения ему имени.

```
boolean itemIsDVD(Sale sale) {  
    if sale.item.equals("DVD") {  
        return true;  
    } else {  
        return false;  
    }  
}  
  
itemIsDVD(sale);
```


Еще пример лямбда-выражения

```
(sale) -> sale.item.equals("DVD")
```

Лямбда-выражение – сжатая запись объявления метода и его вызова без присвоения ему имени.

```
boolean itemIsDVD(Sale sale) {  
    return sale.item.equals("DVD");  
}  
  
itemIsDVD(sale);
```



Еще пример лямбда-выражения

1. Выбрать очередной объект из входного потока. Временно присвоить переменной *sale* ссылку на этот объект.

```
(sale) -> sale.item.equals("DVD")
```

2. Получить значение поля *item* данного объекта *sale*.

3. Проверить, чему равен результат сравнения значения поля *item* объекта *sale* и строки "DVD": *true* или *false*.

Лямбда-выражение получает объекты из потока и вызывает для каждого объекта метод, похожий на метод `itemIsDVD()`.

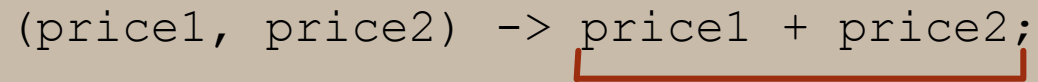
Результат-?

Лямбда-выражение с двумя параметрами

```
(price1, price2) -> price1 + price2;
```

- 1 Выбрать два значения из входного потока. Временно присвоить эти значения переменным price1 и price2.

```
(price1, price2) -> price1 + price2;
```



- 2 Найти сумму этих двух значений.

Лямбда-выражение с двумя параметрами

```
(price1, price2) -> price1 + price2;
```

Лямбда-выражение получает значения из потока и вызывает метод, похожий на метод `sum()`, для суммирования значений.

Результат-?

```
double sum(double price1, double price2){  
    return price1 + price2;  
}
```

```
sum(price1, price2) ;
```

Некоторые виды лямбда-выражений

Название	Описание	Пример
Function	Принимает один параметр; выдает результат любого типа	<code>(sale) -> sale.price</code>
Predicate	Принимает один параметр; выдает результат, оцениваемый как булево значение	<code>(sale) -> sale.item.equals("DVD")</code>
Binary Operator	Принимает два параметра; выдает результат того же типа	<code>(price1, price2) -> price1 + price2</code>
Consumer	Принимает один параметр; не выдает никакого результата	<code>(sale) -> System.out.println(sale.price)</code>

Некоторые методы функционального программирования

Имя метода	Членство	Параметры	Тип результата	Значение результата
<code>stream()</code>	Collection (например, объект <code>ArrayList</code>)	Отсутствуют	Stream	Поток, который извлекает элементы из коллекции
<code>filter()</code>	Stream	Predicate	Stream	Новый поток, содержащий значения, для которых лямбда-выражение возвращает <code>true</code>
<code>map()</code>	Stream	Function	Stream	Новый поток, содержащий результаты применения лямбда выражения ко входному потоку
<code>reduce()</code>	Stream	Binary Operator	Тип, используемый в Bin Operator	Результат сочетания всех значений входного потока

Решение в стиле функционального программирования

Программа вычисления общей суммы продаж DVD.

Использование класса Sale

```
public class TallySales{
    Public static void main(String[] args) {
        ArrayList<Sale> sales = new ArrayList<Sale>();
        NumberFormat currency = NumberFormat.getCurrencyInstance();

        fillTheList(sales);
        System.out.println(currency.format(
            sales.stream()
                .filter((sale) -> sale.item.equals("DVD"))
                .map((sale) -> sale.price)
                .reduce(0.0, (price1, price2) -> price1+price2))));
    }

    Static void fillTheList(ArrayList<Sale> sales) {
        sales.add(new Sale("DVD", 15.00));
        sales.add(new Sale("Книга", 12.00));
        sales.add(new Sale("DVD", 21.00));
        sales.add(new Sale("CD", 5.25));
    }
}
```

Цепочка функционального программирования

Из коллекции sales

...сначала получаем один поток...

...Затем другой поток...

...затем поток значений типа double...

...а затем значение типа double...

```
("DVD",15.00)  
("BOOK",12.00)  
("CD",5.25)  
("DVD",21.00)
```

`.stream()`

```
("DVD",15.00)  
("BOOK",12.00)  
("CD",5.25)  
("DVD",21.00)
```

`.filter((sale)->sale.item.equals("DVD"))`

```
("DVD",15.00)  
("DVD",21.00)
```

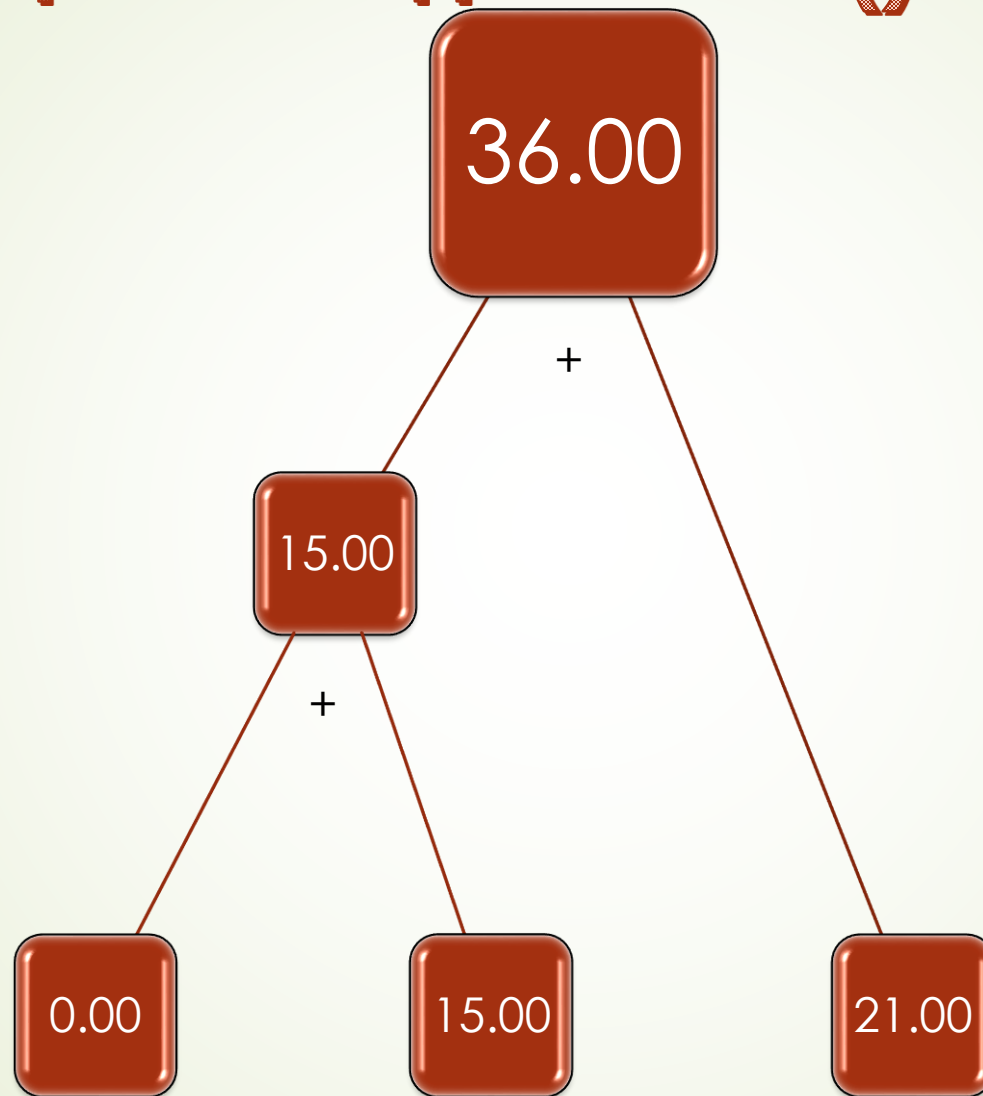
`.map((sale)->sale.price)`

```
15.00 21.00
```

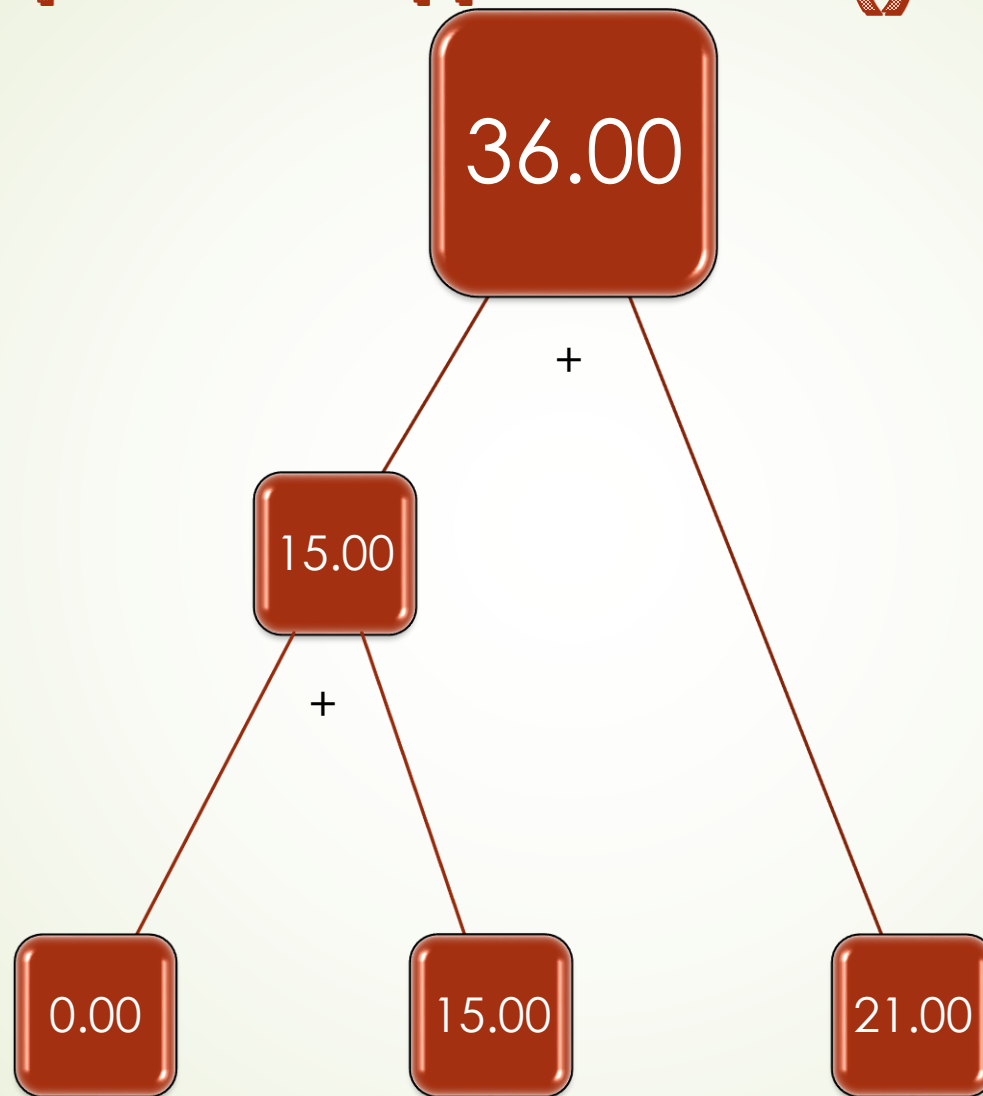
`.reduce(0.0, (price1, price2) -> price1+price2))`

```
36
```

Сложение двух значений из входного потока с помощью метода reduce()



Сложение двух значений из входного потока с помощью метода reduce()



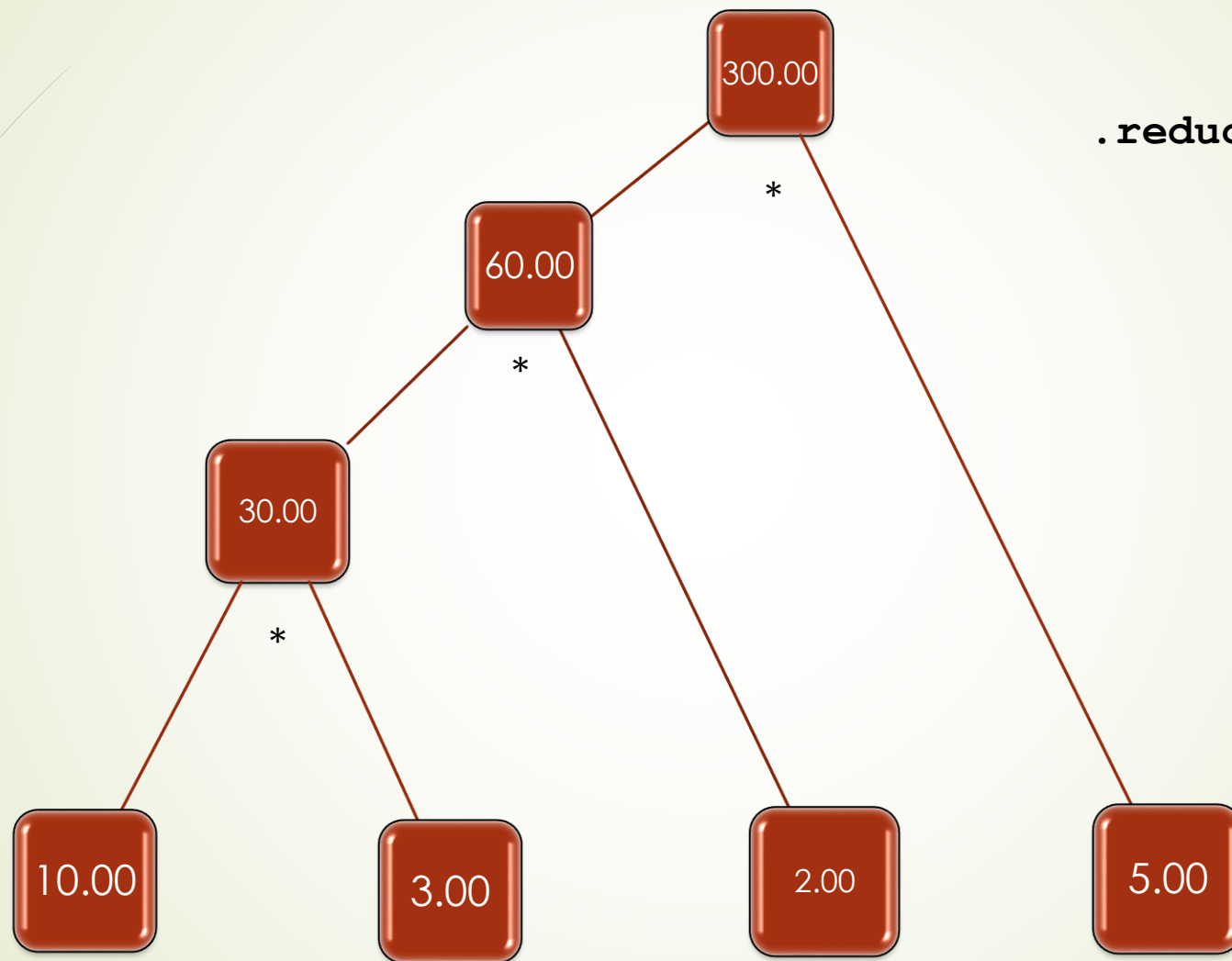
Сложение двух значений из входного потока с помощью метода `reduce()`

Рассмотрим пример, в котором вызов

```
.reduce(10.0, (v1, v2) ->v1*v2)
```

выполняется по отношению к потоку, содержащему значения 3.0, 2.0, 5.0

Сложение двух значений из входного потока с помощью метода `reduce()`



`.reduce(10.0, (v1, v2) ->v1*v2)`