1

название

2

Концепция лямбда-выражений появилась в Java с версии 1.8 и добавила в язык элементы функционального программирования.

В реальных задачах постоянно возникала необходимость передать простое действие: например, что выполнить в новом потоке или как сравнить объекты при сортировке.

Сначала Java предлагала создавать для этого отдельный класс и реализовывать интерфейс. Но ради одной строчки логики писать целый класс — слишком громоздко. Поэтому появились **анонимные классы**: без имени, которые создаются прямо в месте вызова. Они позволяли избежать множества файлов и описывать логику сразу там, где она нужна.

Однако анонимные классы тоже оставались громоздкими. И всё это ради одной-двух строк действий. На слайде видно, что даже простая задача «вывести Привет,мир! в отдельном потоке» требовала целого блока кода.

Эта проблема и привела к появлению **лямбда-выражений**. То же самое стало возможно записать в одну строку. Код стал компактнее, его проще читать и воспринимать. Это сделало Java современнее и стало шагом к более функциональному стилю программирования».

3

Лямбда-выражение представляет собой блок кода, запоминающий контекст вокруг себя в момент создания. Синтаксис лямбда-функции в Java состоит из аргументов и lambda-выражений. Шаблон лямбда-функции выглядит так:

В скобках указываются параметры выражения, дальше идёт лямбда-оператор ->, после которого записывается само выражение или целый блок команд

4

Рассмотрим пример вывода рандомного числа. Так же как и обычная функция, лямбда выражение только запоминает код, но не вызывает. В этом примере рэндом сплайер выполнится только при вызове через гэт. То есть мы определяем в одном месте программы лямбда-выражение и затем можем его вызывать при необходимости неопределенное количество раз в различных частях программы.

5

На этом примере видно, как можно использовать лямбды для задания простых операций.

У нас есть интерфейс IntegerMath с одним методом operation(int a, int b). В классе Calculator мы описываем метод operateBinary, который принимает два числа и реализацию интерфейса.

В main мы создаём две лямбды:

– Первая (addition) записана **однострочно** — (a, b) -> a + b. Такой вариант называется *single-expression body*: когда у нас только одно выражение, фигурные скобки и return можно не писать.

– Вторая (subtraction) оформлена как **блок кода** — (a, b) -> { return a - b; }. Такой вариант нужен, если внутри лямбды больше одного выражения, или мы явно используем return. Это *multi-statement body*.

Дальше мы передаём эти лямбды в метод operateBinary и выполняем операции сложения и вычитания.

Таким образом, на примере видно, что лямбды могут быть как максимально компактными — в одну строку, так и более развёрнутыми — если нужно несколько инструкций.

6

Чтобы понять, как работают лямбды в Java, нужно познакомиться с понятием «функциональный интерфейс». Это интерфейс, у которого есть ровно один абстрактный метод. Его ещё называют SAM — Single Abstract Method.

Лямбда-выражение по сути и есть реализация этого метода. Поэтому лямбда всегда «привязывается» к функциональному интерфейсу.

В этом примере мы через лямбда выражение будем определять различные математические операции: сложение и вычитание. Метод Calculate абстрактный, то есть без реализации. И как раз его реализацию мы передаем через лямбда выражение, когда создаём объекты sum и sub. При вызове метода Calculate мы получаем разные результаты в зависимости от того как создавали соответствующий объект

7

У нас есть список людей List<Person>, и мы хотим выбирать из него только тех, кто подходит по определённым условиям. Посмотрим, как можно решать эту задачу по-разному — и почему в итоге лучше использовать лямбды.

Самый простой способ — написать отдельный метод. Но если критериев много — пол, возраст, регион — придётся писать десятки таких методов. Код становится избыточным и плохо поддерживается.

8

Улучшаем подход: делаем универсальный метод printPersons, который принимает интерфейс CheckPerson. Через тестер можем прописывать разные условия ,в зависимости от задачи. Таким образом мы избавились от большого числа функций и свели отбор только к одной. Условие можно задавать анонимным классом. Но для 1 проверки по прежнему остаётся много кода. Синтаксис перегружен.

9

А теперь используем лямбду. Интерфейс CheckPerson остаётся, но код условия записан одной строкой. Всё лишнее исчезло. Сразу видно саму логику, а не технические детали.

В итоге лямбды позволяют убрать всё лишнее и оставить только главное — **логику**. Именно поэтому они стали важной частью Java.

10

В Java уже есть готовый стандартный интерфейс Predicate<T> с методом test. Мы можем использовать его, и тогда метод printPersonsWithPredicate принимает условие фильтрации в виде лямбды.

Это сильно упрощает код: логика проверки теперь передаётся прямо «на месте».

Именно на этом примере мы подходим к важной теме — Java содержит целый набор таких стандартных функциональных интерфейсов. Среди них есть четыре основные группы: **Predicate, Function, Consumer и Supplier**. Они покрывают самые частые сценарии работы с данными. Давайте рассмотрим их подробнее.

11

* Как мы уже сказали **Predicate** отвечает за фильтрацию. Принимает объект, возвращает boolean.Такие проверки особенно часто применяются в Stream API, когда нам нужно отобрать элементы по условию.

**Function** отвечает за преобразование. Берём один объект, на выходе получаем другой. Разберём небольшой пример: пусть дан массив строк, который преобразуется в stream, затем вызывается метод map(Function mapper), преобразующий поток объектов String в поток чисел-длин строк.

* Интерфейс **Consumer** представляет абстрактный метод void accept(T t), функция, принимающая объект типа T и выполняющая над ним некоторое действие. Результат действия можно сохранить во внешнем объекте, например, коллекции или вывести в поток вывода, например, в файл или на консоль. В следующем примере consumer преобразует строку на основе разделителя в массив строк

Интерфейс **Supplier** возвращает новый объект типа T методом T get(). Предназначен для создания новых объектов. Прямое применение интерфейса для создания объектов выглядит следующим образом.

12

Здесь показано две реализации одной и той же задачи: вывести на экран чётные числа из списка.

В верхней части используется классический подход — цикл for. Мы вручную перебираем элементы, проверяем условие number % 2 == 0, и если оно выполняется, печатаем число. Код работает, но выглядит достаточно многословным.

В нижней части — та же задача решена через поток (stream) и лямбды. Мы создаём поток из списка, фильтруем элементы по условию x -> x % 2 == 0 и сразу же печатаем результат через forEach. Все операторы своим аргументом имеют лямбда выражения.

13

Хотя чаще всего лямбды используют для промежуточных операций в Stream (map, filter, forEach и т.д.), они также активно применяются в Collectors.

Collectors — это набор готовых «сборщиков», которые умеют превращать поток данных (Stream) в разные структуры:

(toList() — собирает в список, toSet() — собирает в множество и тд)

В этом примере Collectors.mapping(...) — это специальный коллектор, который сначала применяет **функцию преобразования** к каждому элементу (у нас лямбда s -> (int) s.charAt(0)), а потом передаёт результат в **другой коллектор** (Collectors.toList()). ;

toList() — это короткая форма.

Ниже пример показывает, что мы можем вручную создать коллекцию через лямбду. toCollection() собирает все элементы потока в коллекцию, которую мы укажем в скобках, то есть в нашем случае в список

**Скобки <>** после имени класса говорят: «внутри этого списка будут элементы такого-то типа».new ArrayList<>() — короткая запись, которая автоматически берёт тип из контекста.

14

Оператор switch в Java используется тогда, когда нужно выбрать один вариант из нескольких. По сути, он заменяет длинные цепочки if-else-if, делая код более аккуратным и читаемым.

Внутри switch мы указываем выражение, а дальше через case описываем возможные варианты. Если совпадение найдено — выполняется соответствующий блок кода. Чтобы выполнение не пошло дальше по остальным вариантам, обычно используется ключевое слово break. Если ни одно условие не подошло, срабатывает блок default.

В более новых версиях Java switch упростили. Вместо символа «:» и return проще использовать лямбда-выражения. Они позволяют сразу возвращать значение или указывать действие в одной строке. А если нужно вернуть результат из блока кода, используется ключевое слово yield. Благодаря этому switch стал не только короче, но и теперь его можно использовать как полноценное выражение.

* Здесь создаётся метод defineLevel, который получает строку role и возвращает уровень доступа в виде числа.
* В старом стиле пришлось бы писать несколько if-else, но switch позволяет компактно перечислить все варианты.
* Если роль "guest", метод вернёт 0; если "admin" — 3 и т. д.
* Если передано неизвестное значение, сработает default: сначала выведется сообщение в консоль, а затем с помощью yield метод вернёт -1.

Оператор yield появился в Java 13 как решение проблемы путаницы с break. В Java 12 ввели switch-выражения, где break использовался одновременно для двух целей: возврата значения и прерывания выполнения. Это двойное назначение было неудобным и неочевидным.

Разработчики жаловались, что такой break нарушает принцип единственной ответственности и усложняет чтение кода. В ответ на эту обратную связь в Java 13 ввели новый оператор yield, который взял на себя только функцию возврата значения.

Теперь yield явно показывает, какое значение возвращает блок case, а break остался для прерывания выполнения в обычных switch-операторах. Это разделение сделало код чище и понятнее.

15

На этом слайде показано понятие **замыкания**. Лямбда-выражения могут «захватывать» переменные из внешнего контекста. В примере у нас есть переменная x = 5, и лямбда () -> System.out.println(x) её использует.

Но есть ограничение: лямбда может работать только с переменными, которые **не меняются после инициализации** — так называемые *effectively final*. Если мы попробуем изменить x позже (x = 6), компилятор выдаст ошибку.

Почему так сделано? Если бы изменения разрешались, то возникла бы неоднозначность: что именно должна напечатать лямбда — старое значение 5 или новое 6? Чтобы избежать путаницы, Java запрещает изменять такие локальные переменные после того, как их захватила лямбда.

16

На слайде показано, что замыкание в Java фиксирует переменные именно в момент создания, а не во время вызова.

Слева:

Lambda напрямую использует переменную x из метода main. В таком случае компилятор требует, чтобы эта переменная была \*effectively final\* — то есть не менялась после того, как её захватила lambda. Поэтому при попытке сделать x = 6 возникает ошибка.

Справа:

Переменная x передаётся в метод как аргумент. Там она копируется в новый параметр value, и именно его захватывает lambda. Переменная x в main и параметр value — это разные переменные. Поэтому x можно спокойно изменить, а замыкание всё равно будет хранить значение, которое было передано (в нашем случае — 5). Поэтому на экран выводится именно 5.

Вывод:

Замыкание в Java «захватывает» значение переменной на момент создания lambda, а не на момент вызова метода run().

17

Рассмотрим распространённый пример: класс счётчик, который считает количество вызовов. Внутри него в методе инкремент есть локальная переменная и лямбда выражение, которое пытается её увеличить. Здесь мы столкнёмся с ошибкой компиляции, потому что пытаемся изменить локальную переменную.

18

Данное ограничение можно обойти если вместо локальной переменной использовать объект. Сейчас мы создали поле count в классе счётчик, теперь лямбда выражение захватывает поле объекта, а это ссылка. И при переходе по постоянной ссылке на удастся изменить состояние объекта.

19

– В обычной Java-функции мы можем передавать несколько параметров сразу, например три числа.

– Но в функциональном стиле можно переписать её так, что функция будет принимать по одному параметру за раз и на каждом шаге возвращать новую функцию, которая ждёт следующий параметр.

– Такой приём называется каррирование.

– Это удобно, потому что можно заранее «зафиксировать» часть параметров и получить заготовку функции, которую используем повторно.

метод берёт три числа и возвращает их произведение

20

В методе main я показываю два способа решить одну и ту же задачу — умножить три числа.

Сначала идёт обычный вызов метода: сразу передаю все три аргумента и получаю результат. Всё просто и привычно.

А ниже — каррированная версия. Здесь я подаю аргументы не все сразу, а по одному. На первом шаге фиксируется 1, на втором шаге — 2, и только на третьем шаге вычисляется результат.

В итоге и там, и там ответ одинаковый — 6, но подходы разные. Обычный метод даёт всё сразу, а каррирование позволяет разбить процесс на шаги и, при необходимости, зафиксировать часть аргументов и переиспользовать функцию.

curry() возвращает функцию, которая ждёт первый аргумент.

.apply(1) → мы фиксируем f = 1, возвращается новая функция.

.apply(2) → фиксируем второй аргумент s = 2, опять получаем функцию.

.apply(3) → подставляем третий аргумент, и на этом шаге вычисляется произведение.