**Что такое ООП?**

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы;

каждый объект является экземпляром определенного класса

классы образуют иерархии.

Программа считается объектно-ориентированной, только если выполнены все три указанных требования. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных.

Согласно парадигме ООП программа состоит из объектов, обменивающихся сообщениями. Объекты могут обладать состоянием, единственный способ изменить состояние объекта - послать ему сообщение, в ответ на которое, объект может изменить собственное состояние.

**Назовите основные принципы ООП.**

Инкапсуляция - сокрытие реализации.

Наследование - создание новой сущности на базе уже существующей.

Полиморфизм - возможность иметь разные формы для одной и той же сущности.

Абстракция - набор общих характеристик.

Посылка сообщений - форма связи, взаимодействия между сущностями.

Переиспользование- все что перечислено выше работает на повторное использование кода.

Это единственно верный порядок парадигм ООП, так как каждая последующая использует предыдущие.

**Что такое «инкапсуляция»?**

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя, открыв только то, что необходимо при последующем использовании.

Цель инкапсуляции — уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса.

Представим на минутку, что мы оказались в конце позапрошлого века, когда Генри Форд ещё не придумал конвейер, а первые попытки создать автомобиль сталкивались с критикой властей по поводу того, что эти коптящие монстры загрязняют воздух и пугают лошадей. Представим, что для управления первым паровым автомобилем необходимо было знать, как устроен паровой котёл, постоянно подбрасывать уголь, следить за температурой, уровнем воды. При этом для поворота колёс использовать два рычага, каждый из которых поворачивает одно колесо в отдельности. Думаю, можно согласиться с тем, что вождение автомобиля того времени было весьма неудобным и трудным занятием.

Теперь вернёмся в сегодняшний день к современным чудесам автопрома с коробкой-автоматом. На самом деле, по сути, ничего не изменилось. Бензонасос всё так же поставляет бензин в двигатель, дифференциалы обеспечивают поворот колёс на различающиеся углы, коленвал превращает поступательное движение поршня во вращательное движение колёс. Прогресс в другом. Сейчас все эти действия скрыты от пользователя и позволяют ему крутить руль и нажимать на педаль газа, не задумываясь, что в это время происходит с инжектором, дроссельной заслонкой и распредвалом. Именно сокрытие внутренних процессов, происходящих в автомобиле, позволяет эффективно его использовать даже тем, кто не является профессионалом-автомехаником с двадцатилетним стажем. Это сокрытие в ООП носит название инкапсуляции.

Пример:

public class SomePhone {

private int year;

private String company;

public SomePhone(int year, String company) {

this.year = year;

this.company = company;

}

private void openConnection(){

//findComutator

//openNewConnection...

}

public void call() {

openConnection();

System.out.println("Вызываю номер");

}

public void ring() {

System.out.println("Дзынь-дзынь");

}

}

Модификатор private делает доступными поля и методы класса только внутри данного класса. Это означает, что получить доступ к private полям из вне невозможно, как и нет возможности вызвать private методы.

Сокрытие доступа к методу openConnection, оставляет нам также возможность к свободному изменению внутренней реализации этого метода, так как этот метод гарантированно не используется другими объектами и не нарушит их работу.

Для работы с нашим объектом мы оставляем открытыми методы call и ring с помощью модификатора public. Предоставление открытых методов для работы с объектом также является частью механизма инкапсуляции, так как если полностью закрыть доступ к объекту – он станет бесполезным.

**Что такое «наследование»?**

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется предком, базовым или родительским. Новый класс – потомком, наследником или производным классом.

Представим себя, на минуту, инженерами автомобильного завода. Нашей задачей является разработка современного автомобиля. У нас уже есть предыдущая модель, которая отлично зарекомендовала себя в течение многолетнего использования. Всё бы хорошо, но времена и технологии меняются, а наш современный завод должен стремиться повышать удобство и комфорт выпускаемой продукции и соответствовать современным стандартам.

Нам необходимо выпустить целый модельный ряд автомобилей: седан, универсал и малолитражный хэтч-бэк. Очевидно, что мы не собираемся проектировать новый автомобиль с нуля, а, взяв за основу предыдущее поколение, внесём ряд конструктивных изменений. Например, добавим гидроусилитель руля и уменьшим зазоры между крыльями и крышкой капота, поставим противотуманные фонари. Кроме того, в каждой модели будет изменена форма кузова.

Очевидно, что все три модификации будут иметь большинство свойств прежней модели (старый добрый двигатель 1970 года, непробиваемая ходовая часть, зарекомендовавшая себя отличным образом на отечественных дорогах, коробку передач и т.д.). При этом каждая из моделей будет реализовать некоторую новую функциональность или конструктивную особенность. В данном случае, мы имеем дело с наследованием.

Пример: Рассмотрим пример создания класса смартфон с помощью наследования. Все беспроводные телефоны работают от аккумуляторных батарей, которые имеют определенный ресурс работы в часах. Поэтому добавим это свойство в класс беспроводных телефонов:

public abstract class WirelessPhone extends AbstractPhone {

private int hour;

public WirelessPhone(int year, int hour) {

super(year);

this.hour = hour;

}

}

Сотовые телефоны наследуют свойства беспроводного телефона, мы также добавили в этот класс реализацию методов call и ring:

public class CellPhone extends WirelessPhone {

public CellPhone(int year, int hour) {

super(year, hour);

}

@Override

public void call(int outputNumber) {

System.out.println("Вызываю номер " + outputNumber);

}

@Override

public void ring(int inputNumber) {

System.out.println("Вам звонит абонент " + inputNumber);

}

}

И, наконец, класс смартфон, который в отличие от классических сотовых телефонов имеет полноценную операционную систему. В смартфон можно добавлять новые программы, поддерживаемые данной операционной системой, расширяя, таким образом, его функциональность. С помощью кода класс можно описать так:

public class Smartphone extends CellPhone {

private String operationSystem;

public Smartphone(int year, int hour, String operationSystem) {

super(year, hour);

this.operationSystem = operationSystem;

}

public void install(String program){

System.out.println("Устанавливаю " + program + "для" + operationSystem);

}

}

Как видите, для описания класса Smartphone мы создали совсем немного нового кода, но получили новый класс с новой функциональностью. Использование этого принципа ООП java позволяет значительно уменьшить объем кода, а значит, и облегчить работу программисту.

**Что такое «полиморфизм»?**

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор языка программирования. Отсюда следует ключевая особенность полиморфизма - использование объекта производного класса, вместо объекта базового (потомки могут изменять родительское поведение, даже если обращение к ним будет производиться по ссылке родительского типа).

Любое обучение вождению не имело бы смысла, если бы человек, научившийся водить, скажем, ВАЗ 2106 не мог потом водить ВАЗ 2110 или BMW X3. С другой стороны, трудно представить человека, который смог бы нормально управлять автомобилем, в котором педаль газа находится левее педали тормоза, а вместо руля – джойстик.

Всё дело в том, что основные элементы управления автомобиля имеют одну и ту же конструкцию, и принцип действия. Водитель точно знает, что для того, чтобы повернуть налево, он должен повернуть руль, независимо от того, есть там гидроусилитель или нет. Если человеку надо доехать с работы до дома, то он сядет за руль автомобиля и будет выполнять одни и те же действия, независимо от того, какой именно тип автомобиля он использует. По сути, можно сказать, что все автомобили имеют один и тот же интерфейс, а водитель, абстрагируясь от сущности автомобиля, работает именно с этим интерфейсом. Если водителю предстоит ехать по немецкому автобану, он, вероятно выберет быстрый автомобиль с низкой посадкой, а если предстоит возвращаться из отдалённого маральника в Горном Алтае после дождя, скорее всего, будет выбран УАЗ с армейскими мостами. Но, независимо от того, каким образом будет реализовываться движение и внутреннее функционирование машины, интерфейс останется прежним.

Полиморфная переменная, это переменная, которая может принимать значения разных типов, а полиморфная функция, это функция, у которой хотя бы один аргумент является полиморфной переменной. Выделяют два вида полиморфных функций:

ad hoc, функция ведет себя по разному для разных типов аргументов (например, функция draw() — рисует по разному фигуры разных типов);

параметрический, функция ведет себя одинаково для аргументов разных типов (например, функция add() — одинаково кладет в контейнер элементы разных типов).

Принцип в ООП, когда программа может использовать объекты с одинаковым интерфейсом без информации о внутреннем устройстве объекта, называется полиморфизмом.

Пример:

Давайте представим, что нам в программе нужно описать пользователя, который может пользоваться любыми моделями телефона, чтобы позвонить другому пользователю. Вот как можно это сделать:

public class User {

private String name;

public User(String name) {

this.name = name;

}

public void callAnotherUser(int number, AbstractPhone phone) {

// вот он полиморфизм - использование в коде абстактного типа AbstractPhone phone!

phone.call(number);

}

}

Теперь опишем различные модели телефонов. Одна из первых моделей телефонов:

public class ThomasEdisonPhone extends AbstractPhone {

public ThomasEdisonPhone(int year) {

super(year);

}

@Override

public void call(int outputNumber) {

System.out.println("Вращайте ручку");

System.out.println("Сообщите номер абонента, сэр");

}

@Override

public void ring(int inputNumber) {

System.out.println("Телефон звонит");

}

}

Обычный стационарный телефон:

public class Phone extends AbstractPhone {

public Phone(int year) {

super(year);

}

@Override

public void call(int outputNumber) {

System.out.println("Вызываю номер" + outputNumber);

}

@Override

public void ring(int inputNumber) {

System.out.println("Телефон звонит");

}

}

И, наконец, крутой видеотелефон:

public class VideoPhone extends AbstractPhone {

public VideoPhone(int year) {

super(year);

}

@Override

public void call(int outputNumber) {

System.out.println("Подключаю видеоканал для абонента " + outputNumber);

}

@Override

public void ring(int inputNumber) {

System.out.println("У вас входящий видеовызов..." + inputNumber);

}

}

Создадим объекты в методе main() и протестируем метод callAnotherUser:

AbstractPhone firstPhone = new ThomasEdisonPhone(1879);

AbstractPhone phone = new Phone(1984);

AbstractPhone videoPhone=new VideoPhone(2018);

User user = new User("Андрей");

user.callAnotherUser(224466,firstPhone);

// Вращайте ручку

//Сообщите номер абонента, сэр

user.callAnotherUser(224466,phone);

//Вызываю номер 224466

user.callAnotherUser(224466,videoPhone);

//Подключаю видеоканал для абонента 224466

Используя вызов одного и того же метода объекта user, мы получили различные результаты. Выбор конкретной реализации метода call внутри метода callAnotherUser производился динамически на основании конкретного типа вызывающего его объекта в процессе выполнения программы. В этом и заключается основное преимущество полиморфизма – выбор реализации в процессе выполнения программы.

В примерах классов телефонов, приведенных выше, мы использовали переопределение методов – причем, при котором изменяется реализация метода, определенная в базовом классе, без изменения сигнатуры метода. По сути, это является заменой метода, и именно новый метод, определенный в подклассе, вызывается при выполнении программы.

Обычно, при переопределении метода, используется аннотация @Override, которая подсказывает компилятору о необходимости проверить сигнатуры переопределяемого и переопределяющего методов.

**Что такое «абстракция»?**

Абстрагирование – это способ выделить набор общих характеристик объекта, исключая из рассмотрения частные и незначимые. Соответственно, абстракция – это набор всех таких характеристик.

Представьте, что водитель едет в автомобиле по оживлённому участку движения. Понятно, что в этот момент он не будет задумываться о химическом составе краски автомобиля, особенностях взаимодействия шестерёнок в коробке передач или влияния формы кузова на скорость (разве что, автомобиль стоит в глухой пробке и водителю абсолютно нечем заняться). Однако, руль, педали, указатель поворота он будет использовать регулярно.

Пример:

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

}

class MyMainClass {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

**Что представляет собой «обмен сообщениями»?**

Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. В ООП посылка сообщения (вызов метода) — это единственный путь передать управление объекту. Если объект должен «отвечать» на это сообщение, то у него должна иметься соответствующий данному сообщению метод. Так же объекты, используя свои методы, могут и сами посылать сообщения другим объектам. Обмен сообщениями реализуется с помощью динамических вызовов, что приводит к чрезвычайно позднему связыванию (extreme late binding).

Пусть требуется создать физическую модель, описывающую сталкивающиеся шары разных размеров. Традиционный подход к решению этой задачи примерно таков: определяется набор данных, описывающих каждый шар (например, его координаты, массу и ускорение); каждому шару присваивается уникальный идентификатор (например, организуется массив, значение индекса которого соответствует номеру шара), который позволит отличать каждый из шаров от всех других. Наконец, пишется подпрограмма с названием, скажем, bounce; эта процедура должна на основе номера шара и его начальных параметров соответствующим образом изменять данные, описывающие шар. В отличие от традиционного подхода объектно-ориентированная версия программы моделирует каждый из шаров посредством объекта. При этом объект, соответствующий конкретному шару, содержит не только его параметры, но и весь код, описывающий поведение шара при различных взаимодействиях. Так, каждый шар будет иметь собственный метод bounce(). Вместо того, чтобы вызывать подпрограмму bounce с аргументом, определяющим, скажем, шар №3, необходимо будет передать объекту «шар №3» сообщение, предписывающее ему выполнить столкновение.

**Расскажите про основные понятия ООП: «класс», «объект», «интерфейс».**

Класс – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт).

С точки зрения программирования класс можно рассматривать как набор данных (полей, атрибутов, членов класса) и функций для работы с ними (методов).

С точки зрения структуры программы, класс является сложным типом данных.

Объект (экземпляр) – это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом. Каждый объект имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе.

Интерфейс – это набор методов класса, доступных для использования. Интерфейсом класса будет являться набор всех его публичных методов в совокупности с набором публичных атрибутов. По сути, интерфейс специфицирует класс, чётко определяя все возможные действия над ним.

**В чем заключаются преимущества и недостатки объектно-ориентированного подхода в программировании?**

Преимущества:

Объектная модель вполне естественна, поскольку в первую очередь ориентирована на человеческое восприятие мира, а не на компьютерную реализацию.

Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.

Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.

Инкапсуляция позволяет привнести свойство модульности, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.

Возможность создавать расширяемые системы.

Использование полиморфизма оказывается полезным при:

Обработке разнородных структур данных. Программы могут работать, не различая вида объектов, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.

Изменении поведения во время исполнения. На этапе исполнения один объект может быть заменен другим, что позволяет легко, без изменения кода, адаптировать алгоритм в зависимости от того, какой используется объект.

Реализации работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом объектов.

Возможности описать независимые от приложения части предметной области в виде набора универсальных классов, или фреймворка, который в дальнейшем будет расширен за счет добавления частей, специфичных для конкретного приложения.

Повторное использование кода:

Сокращается время на разработку, которое может быть отдано другим задачам.

Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.

Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с ним программ.

Если программа опирается на стандартные компоненты, ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает использование.

Недостатки:

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу.

Код для обработки сообщения иногда «размазан» по многим методам (иначе говоря, обработка сообщения требует не одного, а многих методов, которые могут быть описаны в разных классах).

Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но и о том, в каком контексте он вызывается.

Неэффективность и неэкономное распределения памяти на этапе выполнения (по причине издержек на динамическое связывание и проверки типов на этапе выполнения).

Излишняя универсальность. Часто содержится больше методов, чем это реально необходимо текущей программе. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом.

**Что подразумевают в плане принципов ООП выражения «является» и «имеет»?**

«является» подразумевает наследование. «имеет» подразумевает ассоциацию (агрегацию или композицию).

**В чем разница между композицией и агрегацией?**

Ассоциация обозначает связь между объектами. Композиция и агрегация — частные случаи ассоциации «часть-целое».

Агрегация предполагает, что объекты связаны взаимоотношением «part-of» (часть). Композиция более строгий вариант агрегации. Дополнительно к требованию «part-of» накладывается условие, что экземпляр «части» может входить только в одно целое (или никуда не входить), в то время как в случае агрегации экземпляр «части» может входить в несколько целых.

Например, книга состоит из страниц, и мы не можем вырвать страницу из книги и вложить в другую книгу. Страницы четко привязаны к конкретной книге, поэтому это композиция. В тоже время мы можем взять и перенести книгу из одной библиотеки в другую - это уже агрегация.

**Что такое статическое и динамическое связывание?**

Присоединение вызова метода к телу метода называется связыванием. Если связывание проводится компилятором (компоновщиком) перед запуском программы, то оно называется статическим или ранним связыванием (early binding).

В свою очередь, позднее связывание (late binding) это связывание, проводимое непосредственно во время выполнения программы, в зависимости от типа объекта. Позднее связывание также называют динамическим (dynamic) или связыванием на стадии выполнения (runtime binding). В языках, реализующих позднее связывание, должен существовать механизм определения фактического типа объекта во время работы программы, для вызова подходящего метода. Иначе говоря, компилятор не знает тип объекта, но механизм вызова методов определяет его и вызывает соответствующее тело метода. Механизм позднего связывания зависит от конкретного языка, но нетрудно предположить, что для его реализации в объекты должна включаться какая-то дополнительная информация.

Для всех методов Java используется механизм позднего (динамического) связывания, если только метод не был объявлен как final (приватные методы являются final по умолчанию).

За что отвечает JVM:

Загрузка, проверка и исполнение байт кода;

Предоставление среды выполнения для выполнения байт-кода;

Управление памятью и очисткой мусора (Garbage collection);

Виртуальная машина Java (Java Virtual Machine) - это механизм, предоставляющий среду выполнения для управления Java-кодом или приложениями. Виртуальная машина является независимой оболочкой исполнения кода, благодаря которой возможен её запуск на любой ОС, без влияния ОС на выполняемую программу.

JVM работает с 2мя типами данных: примитивные типы (primitive types) и ссылочные типы (reference types).

Примитивы

JVM работает с примитивными значениями (целыми числами и числами с плавающей точкой). По сути, JVM - это 32-битная машина. Типы long и double, которые являются 64-битными, поддерживаются изначально, но занимают две единицы памяти в frame's local или стеке операндов, поскольку каждая единица составляет 32 бита. Типы boolean, byte, short и char имеют расширенный знак (кроме char с нулевым расширением) и работают как 32-разрядные целые числа, так же, как и типы int. Меньшие типы имеют только несколько специфических для типа инструкций для загрузки, хранения и преобразования типов. boolean значение работает как 8-битное byte значения, где 0 представляет значение false, а 1 - значение true.

Типы ссылок и значения

Существует три типа ссылочных типов: типы классов, типы массивов и типы интерфейсов. Их значения являются ссылками на динамически создаваемые экземпляры классов, массивы или экземпляры классов, которые реализуют интерфейсы соответственно.

Classloader

Загрузчик классов является частью JRE, которая динамически загружает Java классы в JVM. Обычно классы загружаются только по запросу. Система исполнения в Java не должна знать о файлах и файловых системах благодаря загрузчику классов. Делегирование является важной концепцией, которую выполняет загрузчик. Загрузчик классов отвечает за поиск библиотек, чтение их содержимого и загрузку классов, содержащихся в библиотеках. Эта загрузка обычно выполняется «по требованию», поскольку она не происходит до тех пор, пока программа не вызовет класс. Класс с именем может быть загружен только один раз данным загрузчиком классов.

При запуске JVM, используются три загрузчика классов:

Bootstrap class loader (Загрузчик класса Bootstrap)

Extensions class loader (Загрузчик класса расширений)

System class loader (Системный загрузчик классов)

Загрузчик класса Bootstrap загружает основные библиотеки Java, расположенные в папке <JAVA\_HOME>/jre/lib. Этот загрузчик является частью ядра JVM, написан на нативном коде.

Загрузчик класса расширений загружает код в каталоги расширений (<JAVA\_HOME>/jre/lib/ext, или любой другой каталог, указанный системным свойством java.ext.dirs).

Системный загрузчик загружает код, найденный в java.class.path, который сопоставляется с переменной среды CLASSPATH. Это реализуется классом sun.misc.Launcher$AppClassLoader.

Загрузчик классов выполняет три основных действия в строгом порядке:

Загрузка: находит и импортирует двоичные данные для типа.

Связывание: выполняет проверку, подготовку и (необязательно) разрешение.

Проверка: обеспечивает правильность импортируемого типа.

Подготовка: выделяет память для переменных класса и инициализация памяти значениями по умолчанию.

Разрешение: преобразует символические ссылки из типа в прямые ссылки.

Инициализация: вызывает код Java, который инициализирует переменные класса их правильными начальными значениями.

Пользовательский загрузчик классов

Загрузчик классов написан на Java. Поэтому возможно создать свой собственный загрузчик классов, не понимая тонких деталей JVM. У каждого загрузчика классов Java есть родительский загрузчик классов, определенный при создании экземпляра нового загрузчика классов или в качестве системного загрузчика классов по умолчанию для виртуальной машины.

Что делает возможным следующее:

загружать или выгружать классы во время выполнения (например, динамически загружать библиотеки во время выполнения, даже из ресурса HTTP). Это важная особенность для:

реализация скриптовых языков;

использование bean builders;

добавить пользовательскую расширение;

позволяя нескольким пространствам имен общаться. Например, это одна из основ протоколов CORBA / RMI;

изменить способ загрузки байт-кода (например, можно использовать зашифрованный байт-код класса Java);

модифицировать загруженный байт-код (например, для переплетения аспектов во время загрузки при использовании аспектно-ориентированного программирования);

Области данных времени выполнения

Run-Time Data Areas. JVM выделяет множество областей данных во время выполнения, которые используются во время выполнения программы. Некоторые участки данных созданы JVM во время старта и уничтожаются во время её выключения. Другие создаются для каждого потока и уничтожаются, когда поток уничтожается.

The pc Register (PCR)

Виртуальная машина Java может поддерживать много потоков исполнения одновременно. Каждый поток виртуальной машины Java имеет свой собственный регистр PC (program counter). В любой момент каждый поток виртуальной машины Java выполняет код одного метода, а именно текущий метод для этого потока. Если этот метод не является native, регистр pc содержит адрес инструкции виртуальной машины Java, выполняемой в настоящее время.

Коротко говоря: для одного потока, существует один PCR, который создается при запуске потока. PCR хранит адрес выполняемой сейчас инструкции JVM.

Java Virtual Machine Stacks

Каждый поток в JVM имеет собственный стек, созданный одновременно с потоком. Стек в JVM хранит frames. Cтеки в JVM могут иметь фиксированный размер или динамически расширяться и сжиматься в соответствии с требованиями вычислений.

Heap

JVM имеет heap (кучу), которая используется всеми потоками виртуальной машины Java. Куча - это область данных времени выполнения, из которой выделяется память для всех экземпляров и массивов классов. Куча создается при запуске виртуальной машины. Хранилище для объектов восстанавливается автоматической системой управления данными (известной как сборщик мусора); объекты никогда не освобождаются явно. JVM не предполагает какого-либо конкретного типа системы автоматического управления хранением данных, и метод управления может быть выбран в соответствии с системными требованиями разработчика. Куча может иметь фиксированный размер или может быть расширена в соответствии с требованиями вычислений и может быть сокращена, если большая куча становится ненужной. Память для кучи не должна быть смежной.

Method Area

JVM имеет область методов, которая является общей для всех потоков. Она хранит структуры для каждого класса, такие как пул констант, данные полей и методов, а также код для методов и конструкторов, включая специальные методы, используемые при инициализации классов и экземпляров, и инициализации интерфейса. Хотя область метода является логически частью кучи, простые реализации могут не обрабатываться сборщиком мусора. Область метода может иметь фиксированный размер или может быть расширена в соответствии с требованиями вычислений и может быть сокращена, если большая область метода становится ненужной.

Run-Time Constant Pool

A run-time constant pool существует для каждого класса или интерфейса в рантайме и представленно constant\_pool таблицей в \*.class файле. Он содержит несколько видов констант: от числовых литералов, известных во время компиляции, до ссылок на методы и поля, которые должны быть разрешены во время выполнения. Сам run-time constant pool выполняет функцию, аналогичную функции таблицы символов для обычного языка программирования, хотя он содержит более широкий диапазон данных, чем типичная таблица символов. Каждый run-time constant pool отделён от JVM's method area. JVM создаёт run-time constant pool вместе с созданием class или interface.

Native Method Stacks

Реализация виртуальной машины Java может использовать обычные стеки, обычно называемые «стеки Си», для поддержки native methods (методов, написанных на языке, отличном от языка программирования Java).

Frames

Frame используется для хранения данных и частичных результатов, а также для выполнения динамического связывания, возврата значений для методов и отправки исключений. Новый frame создается каждый раз, когда вызывается метод. Frame уничтожается, когда завершается вызов метода, является ли это завершение нормальным или резким (он генерирует неперехваченное исключение). Frames выделяются из стека потока, создающего frame. Каждый frame имеет свой собственный массив локальных переменных, свой собственный стек операндов и ссылку на пул констант во время выполнения класса текущего метода. Размеры массива локальных переменных и стека операндов определяются во время компиляции и предоставляются вместе с кодом для метода, связанного с фреймом. Таким образом, размер структуры данных, frame-а зависит только от реализации виртуальной машины Java, и память для этих структур может быть выделена одновременно при вызове метода.

Только один frame активен в любой точке данного потока управления - метода выполнения, и это frame называется текущим, а его метод известен как текущий метод. Класс, в котором определен текущий метод, является текущим классом. Операции над локальными переменными и стеком операндов обычно выполняются со ссылкой на текущий frame.

Frame перестает быть текущим, если его метод вызывает другой метод или если его метод завершается. Когда метод вызывается, новый frame создается и становится текущим, когда управление переходит к новому методу. При возврате метода текущий frame передает результат вызова метода, если таковой имеется, в предыдущий frame. Текущий frame затем отбрасывается, так как предыдущий frame становится текущим. Обратите внимание, что frame, созданный потоком, является локальным для этого потока и на него не может ссылаться ни один другой поток.

Локальные переменные

Каждый frame содержит массив переменных, известных как его локальные переменные. Длина массива локальных переменных frame определяется во время компиляции и предоставляется в двоичном представлении класса или интерфейса вместе с кодом для метода, связанного с frame-ом. Единичная локальная переменная может хранить значение типа: boolean, byte, char, short, int, float, reference, or returnAddress. Пара локальных переменных может хранить значение типов: long или double.

Локальные переменные адресуются путем индексации. Индекс первой локальной переменной равен нулю.

Значение типа long или типа double занимает две последовательные локальные переменные.

JVM использует локальные переменные для передачи параметров при вызове метода. При вызове метода класса все параметры передаются в последовательных локальных переменных, начиная с локальной переменной 0. При вызове метода экземпляра локальная переменная 0 всегда используется для передачи ссылки на объект, для которого вызывается метод экземпляра (this в Java). Любые параметры впоследствии передаются в последовательных локальных переменных, начиная с локальной переменной 1.

Стеки операндов (Operand Stacks)

Каждый frame содержит стек «последний вошел - первый вышел» (LIFO), известный как стек операндов. Максимальная глубина стека операндов frame-a определяется во время компиляции и предоставляется вместе с кодом для метода, связанного с frame-ом.

Стек операнда пуст при создании frame-a, который его содержит. JVM предоставляет инструкции для загрузки констант или значений из локальных переменных или полей в стек операндов. Другие инструкции JVM берут операнды из стека операндов, оперируют с ними и помещают результат обратно в стек операндов. Стек операндов также используется для подготовки параметров для передачи в методы и для получения результатов метода.

Для примера, инструкция iadd суммирует два int-вых значения. От стека операндов требуется, чтобы два int-вых значения были наверху стека. Значения удаляются из стека, операция pop. Суммируются и их сумма помещается в стек операндов.

Динамическое связывание (Dynamic Linking)

Каждый frame содержит ссылку на run-time constant pool для типа текущего метода для поддержки динамического связывания кода метода. Доступ к вызываемым методам и переменным осуществляется через символические ссылки из class файла. Динамическое связывание преобразует эти символьные ссылки на методы в конкретные ссылки на методы, загружая классы по мере необходимости для разрешения пока еще не определенных символов, и преобразует обращения к переменным в соответствующие смещения в структурах хранения, связанных с расположением этих переменных во время выполнения.

Позднее связывание методов и переменных вносит изменения в другие классы, которые метод использует с меньшей вероятностью нарушить этот код.

Нормальное завершение вызова метода

Вызов метода завершается нормально, если этот вызов не вызывает исключение, либо непосредственно из JVM, либо в результате выполнения явного оператора throw. Если вызов текущего метода завершается нормально, то значение может быть возвращено вызывающему методу. Это происходит, когда вызванный метод выполняет одну из инструкций возврата, выбор которых должен соответствовать типу возвращаемого значения (если оно есть).

Текущий frame используется в этом случае для восстановления состояния инициатора, включая его локальные переменные и стек операндов, с соответствующим образом увеличенным программным счетчиком инициатора, чтобы пропустить инструкцию вызова метода. Затем выполнение обычно продолжается в frame вызывающего метода с возвращенным значением (если оно есть), помещаемым в стек операндов этого frame.

Резкое завершение вызова метода

Вызов метода завершается преждевременно, если при выполнении инструкции JVM в методе выдает исключение, и это исключение не обрабатывается в методе. Выполнение команды throw также приводит к явному выбрасыванию исключения, и, если исключение не перехватывается текущим методом, приводит к неожиданному завершению вызова метода. Вызов метода, который завершается внезапно, никогда не возвращает значение своему вызывающему.

Execution Engine

Байт-код, назначенный run-time data areas, будет выполнен execution engine. Механизм выполнения считывает байт-код и выполняет его по частям.

Interpreter

Интерпретатор интерпретирует байт-код быстро, но выполняется медленно. Недостаток интерпретатора заключается в том, что, когда один метод вызывается несколько раз, каждый раз требуется новая интерпретация.

JIT Compiler

JIT-компилятор устраняет недостатки интерпретатора. Механизм выполнения будет использовать помощь интерпретатора при преобразовании байт-кода, но когда он находит повторный код, он использует JIT-компилятор, который компилирует весь байт-код и изменяет его на собственный код. Этот нативный код будет использоваться непосредственно для повторных вызовов методов, которые улучшают производительность системы.

Генератор промежуточного кода (Intermediate Code Generator). Производит промежуточный код.

Code Optimizer. Отвечает за оптимизацию промежуточного кода, сгенерированного выше.

Генератор целевого кода (Target Code Generator). Отвечает за генерацию машинного кода или родной код.

Профилировщик (Profiler). Специальный компонент, отвечающий за поиск горячих точек, то есть, вызывается ли метод несколько раз или нет.

Garbage Collector

**Чем различаются JRE, JVM и JDK?**

JVM, Java Virtual Machine (Виртуальная машина Java) — основная часть среды времени исполнения Java (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java. JVM может также использоваться для выполнения программ, написанных на других языках программирования.

JRE, Java Runtime Environment (Среда времени выполнения Java) - минимально-необходимая реализация виртуальной машины для исполнения Java-приложений. Состоит из JVM и стандартного набора библиотек классов Java.

JDK, Java Development Kit (Комплект разработки на Java) - JRE и набор инструментов разработчика приложений на языке Java, включающий в себя компилятор Java, стандартные библиотеки классов Java, примеры, документацию, различные утилиты.

Коротко: JDK - среда для разработки программ на Java, включающая в себя JRE - среду для обеспечения запуска Java программ, которая в свою очередь содержит JVM - интерпретатор кода Java программ.

**Какие существуют модификаторы доступа?**

private (приватный): члены класса доступны только внутри класса. Для обозначения используется служебное слово private.

default, package-private, package level (доступ на уровне пакета): видимость класса/членов класса только внутри пакета. Является модификатором доступа по умолчанию - специальное обозначение не требуется.

protected (защищённый): члены класса доступны внутри пакета и в наследниках. Для обозначения используется служебное слово protected.

public (публичный): класс/члены класса доступны всем. Для обозначения используется служебное слово public.

Последовательность модификаторов по возрастанию уровня закрытости: public, protected, default, private.

Во время наследования возможно изменения модификаторов доступа в сторону большей видимости (для поддержания соответствия принципу подстановки Барбары Лисков).

**О чем говорит ключевое слово final?**

Модификатор final может применяться к переменным, параметрам методов, полям и методам класса или самим классам.

Класс не может иметь наследников;

Метод не может быть переопределен в классах наследниках;

Поле не может изменить свое значение после инициализации;

Параметры методов не могут изменять своё значение внутри метода;

Локальные переменные не могут быть изменены после присвоения им значения.

**Какими значениями инициализируются переменные по умолчанию?**

Числа инициализируются 0 или 0.0;

char — \u0000;

boolean — false;

Объекты (в том числе String) — null.

**Что вы знаете о функции main()?**

Метод main() — точка входа в программу. В приложении может быть несколько таких методов. Если метод отсутствует, то компиляция возможна, но при запуске будет получена ошибка `Error: Main method not found`.

public static void main(String[] args) {}

**Какие логические операции и операторы вы знаете?**

&: Логическое AND (И);

&&: Сокращённое AND;

|: Логическое OR (ИЛИ);

||: Сокращённое OR;

^: Логическое XOR (исключающее OR (ИЛИ));

!: Логическое унарное NOT (НЕ);

&=: AND с присваиванием;

|=: OR с присваиванием;

^=: XOR с присваиванием;

==: Равно;

!=: Не равно;

?:: Тернарный (троичный) условный оператор.

**Что такое тернарный оператор выбора?**

Тернарный условный оператор ?: - оператор, которым можно заменить некоторые конструкции операторов if-then-else.

Выражение записывается в следующей форме:

условие ? выражение1 : выражение2

Если условие выполняется, то вычисляется выражение1 и его результат становится результатом выполнения всего оператора. Если же условие равно false, то вычисляется выражение2 и его значение становится результатом работы оператора. Оба операнда выражение1 и выражение2 должны возвращать значение одинакового (или совместимого) типа.

**Какие побитовые операции вы знаете?**

~: Побитовый унарный оператор NOT;

&: Побитовый AND;

&=: Побитовый AND с присваиванием;

|: Побитовый OR;

|=: Побитовый OR с присваиванием;

^: Побитовый исключающее XOR;

^=: Побитовый исключающее XOR с присваиванием;

>>: Сдвиг вправо (деление на 2 в степени сдвига);

>>=: Сдвиг вправо с присваиванием;

>>>: Сдвиг вправо без учёта знака;

>>>=: Сдвиг вправо без учёта знака с присваиванием;

<<: Сдвиг влево (умножение на 2 в степени сдвига);

<<=: Сдвиг влево с присваиванием.

**Где и для чего используется модификатор abstract?**

Класс, помеченный модификатором abstract, называется абстрактным классом. Такие классы могут выступать только предками для других классов. Создавать экземпляры самого абстрактного класса не разрешается. При этом наследниками абстрактного класса могут быть как другие абстрактные классы, так и классы, допускающие создание объектов.

Метод, помеченный ключевым словом abstract - абстрактный метод, т.е. метод, который не имеет реализации. Если в классе присутствует хотя бы один абстрактный метод, то весь класс должен быть объявлен абстрактным.

Использование абстрактных классов и методов позволяет описать некий шаблон объекта, который должен быть реализован в других классах. В них же самих описывается лишь некое общее для всех потомков поведение.

**Дайте определение понятию «интерфейс». Какие модификаторы по умолчанию имеют поля и методы интерфейсов?**

Ключевое слово interface используется для создания полностью абстрактных классов. Основное предназначение интерфейса - определять каким образом мы можем использовать класс, который его реализует. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не реализует их поведение. Все методы неявно объявляются как public.

Начиная с Java 8 в интерфейсах разрешается размещать реализацию методов по умолчанию default и статических static методов.

Интерфейс также может содержать и поля. В этом случае они автоматически являются публичными public, статическими static и неизменяемыми final.

**Чем абстрактный класс отличается от интерфейса? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?**

В Java класс может одновременно реализовать несколько интерфейсов, но наследоваться только от одного класса.

Абстрактные классы используются только тогда, когда присутствует тип отношений «is a» (является). Интерфейсы могут реализоваться классами, которые не связаны друг с другом.

Абстрактный класс - средство, позволяющее избежать написания повторяющегося кода, инструмент для частичной реализации поведения. Интерфейс - это средство выражения семантики класса, контракт, описывающий возможности. Все методы интерфейса неявно объявляются как public abstract или (начиная с Java 8) default - методами с реализацией по-умолчанию, а поля - public static final.

Интерфейсы позволяют создавать структуры типов без иерархии.

Наследуясь от абстрактного, класс «растворяет» собственную индивидуальность. Реализуя интерфейс, он расширяет собственную функциональность.

Абстрактные классы содержат частичную реализацию, которая дополняется или расширяется в подклассах. При этом все подклассы схожи между собой в части реализации, унаследованной от абстрактного класса, и отличаются лишь в части собственной реализации абстрактных методов родителя. Поэтому абстрактные классы применяются в случае построения иерархии однотипных, очень похожих друг на друга классов. В этом случае наследование от абстрактного класса, реализующего поведение объекта по умолчанию может быть полезно, так как позволяет избежать написания повторяющегося кода. Во всех остальных случаях лучше использовать интерфейсы.

**Почему в некоторых интерфейсах вообще не определяют методов?**

Это так называемые маркерные интерфейсы. Они просто указывают что класс относится к определенному типу. Примером может послужить интерфейс Clonable, который указывает на то, что класс поддерживает механизм клонирования.

**Почему нельзя объявить метод интерфейса с модификатором final?**

В случае интерфейсов указание модификатора final бессмысленно, т.к. все методы интерфейсов неявно объявляются как абстрактные, т.е. их невозможно выполнить, не реализовав где-то еще, а этого нельзя будет сделать, если у метода идентификатор final.

**Что имеет более высокий уровень абстракции - класс, абстрактный класс или интерфейс?**

Интерфейс.

**Может ли объект получить доступ к члену класса, объявленному как private? Если да, то каким образом?**

Внутри класса доступ к приватной переменной открыт без ограничений;

Вложенный класс имеет полный доступ ко всем (в том числе и приватным) членам содержащего его класса;

Доступ к приватным переменным извне может быть организован через отличные от приватных методов, которые предоставлены разработчиком класса. Например: getX() и setX().

Через механизм рефлексии (Reflection API):

class Victim {

private int field = 42;

}

//...

Victim victim = new Victim();

Field field = Victim.class.getDeclaredField("field");

field.setAccessible(true);

int fieldValue = (int) field.get(victim);

//...

**Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?**

Сначала вызываются все статические блоки в очередности от первого статического блока корневого предка и выше по цепочке иерархии до статических блоков самого класса.

Затем вызываются нестатические блоки инициализации корневого предка, конструктор корневого предка и так далее вплоть до нестатических блоков и конструктора самого класса.

Parent static block(s) → Child static block(s) → Grandchild static block(s)

→ Parent non-static block(s) → Parent constructor →

→ Child non-static block(s) → Child constructor →

→ Grandchild non-static block(s) → Grandchild constructor

Пример 1:

public class MainClass {

public static void main(String args[]) {

System.out.println(TestClass.v);

new TestClass().a();

}

}

public class TestClass {

public static String v = "Some val";

{

System.out.println("!!! Non-static initializer");

}

static {

System.out.println("!!! Static initializer");

}

public void a() {

System.out.println("!!! a() called");

}

}

Результат выполнения:

!!! Static initializer

Some val

!!! Non-static initializer

!!! a() called

Пример 2:

public class MainClass {

public static void main(String args[]) {

new TestClass().a();

}

}

public class TestClass {

public static String v = "Some val";

{

System.out.println("!!! Non-static initializer");

}

static {

System.out.println("!!! Static initializer");

}

public void a() {

System.out.println("!!! a() called");

}

}

Результат выполнения:

!!! Static initializer

!!! Non-static initializer

!!! a() called

**Зачем нужны и какие бывают блоки инициализации?**

Блоки инициализации представляют собой код, заключенный в фигурные скобки и размещаемый внутри класса вне объявления методов или конструкторов.

Существуют статические и нестатические блоки инициализации.

Блок инициализации выполняется перед инициализацией класса загрузчиком классов или созданием объекта класса с помощью конструктора.

Несколько блоков инициализации выполняются в порядке следования в коде класса.

Блок инициализации способен генерировать исключения, если их объявления перечислены в throws всех конструкторов класса.

Блок инициализации возможно создать и в анонимном классе.

**К каким конструкциям Java применим модификатор static?**

полям;

методам;

вложенным классам;

членам секции import.

**Для чего в Java используются статические блоки инициализации?**

Статические блоки инициализация используются для выполнения кода, который должен выполняться один раз при инициализации класса загрузчиком классов, в момент, предшествующий созданию объектов этого класса при помощи конструктора. Такой блок (в отличие от нестатических, принадлежащих конкретном объекту класса) принадлежит только самому классу (объекту метакласса Class).

**Что произойдёт, если в блоке инициализации возникнет исключительная ситуация?**

Для нестатических блоков инициализации, если выбрасывание исключения прописано явным образом требуется, чтобы объявления этих исключений были перечислены в throws всех конструкторов класса. Иначе будет ошибка компиляции. Для статического блока выбрасывание исключения в явном виде, приводит к ошибке компиляции.

В остальных случаях, взаимодействие с исключениями будет проходить так же, как и в любом другом месте. Класс не будет инициализирован, если ошибка происходит в статическом блоке и объект класса не будет создан, если ошибка возникает в нестатическом блоке.

**Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?**

Если возникшее исключение - наследник RuntimeException:

для статических блоков инициализации будет выброшено java.lang.ExceptionInInitializerError;

для нестатических будет проброшено исключение-источник.

Если возникшее исключение - наследник Error, то в обоих случаях будет выброшено java.lang.Error. Исключение: java.lang.ThreadDeath - смерть потока. В этом случае никакое исключение выброшено не будет.

**Может ли статический метод быть переопределён или перегружен?**

Перегружен - да. Всё работает точно так же, как и с обычными методами - 2 статических метода могут иметь одинаковое имя, если количество их параметров или типов различается.

Переопределён - нет. Выбор вызываемого статического метода происходит при раннем связывании (на этапе компиляции, а не выполнения) и выполняться всегда будет родительский метод, хотя синтаксически переопределение статического метода - это вполне корректная языковая конструкция.

В целом, к статическим полям и методам рекомендуется обращаться через имя класса, а не объект.

**Могут ли нестатические методы перегрузить статические?**

Да. В итоге получится два разных метода. Статический будет принадлежать классу и будет доступен через его имя, а нестатический будет принадлежать конкретному объекту и доступен через вызов метода этого объекта.

**Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?**

При переопределении метода нельзя сузить модификатор доступа к методу (например с public в MainClass до private в Class extends MainClass).

Изменить тип возвращаемого значения при переопределении метода нельзя, будет ошибка attempting to use incompatible return type.

Можно сузить возвращаемое значение, если они совместимы.

Например:

public class Animal {

public Animal eat() {

System.out.println("animal eat");

return null;

}

public Long calc() {

return null;

}

}

public class Dog extends Animal {

public Dog eat() {

return new Dog();

}

/\*attempting to use incompatible return type

public Integer calc() {

return null;

}

\*/

}

**Возможно ли при переопределении метода изменить: модификатор доступа, возвращаемый тип, тип аргумента или их количество, имена аргументов или их порядок; убирать, добавлять, изменять порядок следования элементов секции throws?**

При переопределении метода сужать модификатор доступа не разрешается, т.к. это приведёт к нарушению принципа подстановки Барбары Лисков. Расширение уровня доступа возможно.

Можно изменять все, что не мешает компилятору понять какой метод родительского класса имеется в виду:

Изменять тип возвращаемого значения при переопределении метода разрешено только в сторону сужения типа (вместо родительского класса - наследника).

При изменении типа, количества, порядка следования аргументов вместо переопределения будет происходить overloading (перегрузка) метода.

Секцию throws метода можно не указывать, но стоит помнить, что она остаётся действительной, если уже определена у метода родительского класса. Так же, возможно добавлять новые исключения, являющиеся наследниками от уже объявленных или исключения RuntimeException. Порядок следования таких элементов при переопределении значения не имеет.

**Как получить доступ к переопределенным методам родительского класса?**

С помощью ключевого слова super мы можем обратиться к любому члену родительского класса - методу или полю, если они не определены с модификатором private.

super.method();

**Можно ли объявить метод абстрактным и статическим одновременно?**

Нет. В таком случае компилятор выдаст ошибку: "Illegal combination of modifiers: ‘abstract’ and ‘static’". Модификатор abstract говорит, что метод будет реализован в другом классе, а static наоборот указывает, что этот метод будет доступен по имени класса.

**В чем разница между членом экземпляра класса и статическим членом класса?**

Модификатор static говорит о том, что данный метод или поле принадлежат самому классу и доступ к ним возможен даже без создания экземпляра класса. Поля, помеченные static инициализируются при инициализации класса. На методы, объявленные как static, накладывается ряд ограничений:

Они могут вызывать только другие статические методы.

Они должны осуществлять доступ только к статическим переменным.

Они не могут ссылаться на члены типа this или super.

В отличии от статических, поля экземпляра класса принадлежат конкретному объекту и могут иметь разные значения для каждого. Вызов метода экземпляра возможен только после предварительного создания объекта класса.

Пример:

public class MainClass {

public static void main(String args[]) {

System.out.println(TestClass.v);

new TestClass().a();

System.out.println(TestClass.v);

}

}

public class TestClass {

public static String v = "Initial val";

{

System.out.println("!!! Non-static initializer");

v = "Val from non-static";

}

static {

System.out.println("!!! Static initializer");

v = "Some val";

}

public void a() {

System.out.println("!!! a() called");

}

}

Результат:

!!! Static initializer

Some val

!!! Non-static initializer

!!! a() called

Val from non-static

**Где разрешена инициализация статических/нестатических полей?**

Статические поля можно инициализировать при объявлении, в статическом или нестатическом блоке инициализации.

Нестатические поля можно инициализировать при объявлении, в нестатическом блоке инициализации или в конструкторе.

**Какие типы классов бывают в java?**

Top level class (Обычный класс):

Abstract class (Абстрактный класс);

Final class (Финализированный класс).

Interfaces (Интерфейс).

Enum (Перечисление).

Nested class (Вложенный класс):

Static nested class (Статический вложенный класс);

Member inner class (Простой внутренний класс);

Local inner class (Локальный класс);

Anonymous inner class (Анонимный класс).

**Расскажите про вложенные классы. В каких случаях они применяются?**

Класс называется вложенным (Nested class), если он определен внутри другого класса. Вложенный класс должен создаваться только для того, чтобы обслуживать обрамляющий его класс. Если вложенный класс оказывается полезен в каком-либо ином контексте, он должен стать классом верхнего уровня. Вложенные классы имеют доступ ко всем (в том числе приватным) полям и методам внешнего класса, но не наоборот. Из-за этого разрешения использование вложенных классов приводит к некоторому нарушению инкапсуляции.

Существуют четыре категории вложенных классов:

Static nested class (Статический вложенный класс);

Member inner class (Простой внутренний класс);

Local inner class (Локальный класс);

Anonymous inner class (Анонимный класс).

Такие категории классов, за исключением первого, также называют внутренними (Inner class). Внутренние классы ассоциируются не с внешним классом, а с экземпляром внешнего.

Каждая из категорий имеет рекомендации по своему применению. Если вложенный класс должен быть виден за пределами одного метода или он слишком длинный для того, чтобы его можно было удобно разместить в границах одного метода и если каждому экземпляру такого класса необходима ссылка на включающий его экземпляр, то используется нестатический внутренний класс. В случае, если ссылка на обрамляющий класс не требуется - лучше сделать такой класс статическим. Если класс необходим только внутри какого-то метода и требуется создавать экземпляры этого класса только в этом методе, то используется локальный класс. А, если к тому же применение класса сводится к использованию лишь в одном месте и уже существует тип, характеризующий этот класс, то рекомендуется делать его анонимным классом.

**Что такое «статический класс»?**

Это вложенный класс, объявленный с использованием ключевого слова static. К классам верхнего уровня модификатор static неприменим.

Какие существуют особенности использования вложенных классов: статических и внутренних? В чем заключается разница между ними?

Вложенные классы могут обращаться ко всем членам обрамляющего класса, в том числе и приватным.

Для создания объекта статического вложенного класса объект внешнего класса не требуется.

Из объекта статического вложенного класса нельзя обращаться к не статическим членам обрамляющего класса напрямую, а только через ссылку на экземпляр внешнего класса.

Обычные вложенные классы не могут содержать статических методов, блоков инициализации и классов. Статические вложенные классы - могут.

В объекте обычного вложенного класса хранится ссылка на объект внешнего класса. Внутри статической такой ссылки нет. Доступ к экземпляру обрамляющего класса осуществляется через указание .this после его имени. Например: Outer.this.

**Что такое «локальный класс»? Каковы его особенности?**

Local inner class (Локальный класс) - это вложенный класс, который может быть декларирован в любом блоке, в котором разрешается декларировать переменные. Как и простые внутренние классы (Member inner class) локальные классы имеют имена и могут использоваться многократно. Как и анонимные классы, они имеют окружающий их экземпляр только тогда, когда применяются в нестатическом контексте.

Локальные классы имеют следующие особенности:

Видны только в пределах блока, в котором объявлены;

Не могут быть объявлены как private/public/protected или static;

Не могут иметь внутри себя статических объявлений методов и классов, но могут иметь финальные статические поля, проинициализированные константой;

Имеют доступ к полям и методам обрамляющего класса;

Могут обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final.

**Что такое «анонимные классы»? Где они применяются?**

Это вложенный локальный класс без имени, который разрешено декларировать в любом месте обрамляющего класса, разрешающем размещение выражений. Создание экземпляра анонимного класса происходит одновременно с его объявлением. В зависимости от местоположения анонимный класс ведет себя как статический либо как нестатический вложенный класс - в нестатическом контексте появляется окружающий его экземпляр.

Анонимные классы имеют несколько ограничений:

Их использование разрешено только в одном месте программы - месте его создания;

Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться;

Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.

Анонимные классы обычно применяются для:

создания объекта функции (function object), например, реализация интерфейса Comparator;

создания объекта процесса (process object), такого как экземпляры классов Thread, Runnable и подобных;

в статическом методе генерации;

инициализации открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.

**Каким образом из вложенного класса получить доступ к полю внешнего класса?**

Статический вложенный класс имеет прямой доступ только к статическим полям обрамляющего класса.

Простой внутренний класс, может обратиться к любому полю внешнего класса напрямую. В случае, если у вложенного класса уже существует поле с таким же литералом, то обращаться к такому полю следует через ссылку на его экземпляр. Например: Outer.this.field.

**Для чего используется оператор assert?**

Assert (Утверждение) — это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Утверждение может автоматически сигнализировать об обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.

Утверждения существенно упрощают локализацию ошибок в коде. Даже проверка результатов выполнения очевидного кода может оказаться полезной при последующем рефакторинге, после которого код может стать не настолько очевидным и в него может закрасться ошибка.

Обычно утверждения оставляют включенными во время разработки и тестирования программ, но отключают в релиз-версиях программ.

Т.к. утверждения могут быть удалены на этапе компиляции либо во время исполнения программы, они не должны менять поведение программы. Если в результате удаления утверждения поведение программы может измениться, то это явный признак неправильного использования assert. Таким образом, внутри assert нельзя вызывать методы, изменяющие состояние программы, либо внешнего окружения программы.

В Java проверка утверждений реализована с помощью оператора assert, который имеет форму:

assert [Выражение типа boolean]; или assert [Выражение типа boolean] : [Выражение любого типа, кроме void];

Во время выполнения программы в том случае, если поверка утверждений включена, вычисляется значение булевского выражения, и если его результат false, то генерируется исключение java.lang.AssertionError. В случае использования второй формы оператора assert выражение после двоеточия задаёт детальное сообщение о произошедшей ошибке (вычисленное выражение будет преобразовано в строку и передано конструктору AssertionError).

**Что такое Heap и Stack память в Java? Какая разница между ними?**

Heap (куча) используется Java Runtime для выделения памяти под объекты и классы. Создание нового объекта также происходит в куче. Это же является областью работы сборщика мусора. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться из любой части приложения.

Stack (стек) это область хранения данных также находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче. Стек в Java работает по схеме LIFO (Последний-зашел-Первый-вышел)

Различия между Heap и Stack памятью:

Куча используется всеми частями приложения, в то время как стек используется только одним потоком исполнения программы.

Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится лишь ссылка на него. Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче.

Объекты в куче доступны с любой точки программы, в то время как стековая память не может быть доступна для других потоков.

Стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы.

Если память стека полностью занята, то Java Runtime бросает исключение java.lang.StackOverflowError. Если заполнена память кучи, то бросается исключение java.lang.OutOfMemoryError: Java Heap Space.

Размер памяти стека намного меньше памяти в куче.

Из-за простоты распределения памяти, стековая память работает намного быстрее кучи.

Для определения начального и максимального размера памяти в куче используются -Xms и -Xmx опции JVM. Для стека определить размер памяти можно с помощью опции -Xss.

**Верно ли утверждение, что примитивные типы данных всегда хранятся в стеке, а экземпляры ссылочных типов данных в куче?**

Не совсем. Примитивное поле экземпляра класса хранится не в стеке, а в куче. Любой объект (всё, что явно или неявно создаётся при помощи оператора new) хранится в куче.

**Каким образом передаются переменные в методы, по значению или по ссылке?**

В Java параметры всегда передаются только по значению, что определяется как «скопировать значение и передать копию». С примитивами это будет копия содержимого. Со ссылками - тоже копия содержимого, т.е. копия ссылки. При этом внутренние члены ссылочных типов через такую копию изменить возможно, а вот саму ссылку, указывающую на экземпляр - нет.

**Для чего нужен сборщик мусора?**

Сборщик мусора (Garbage Collector) должен делать всего две вещи:

Находить мусор - неиспользуемые объекты. (Объект считается неиспользуемым, если ни одна из сущностей в коде, выполняемом в данный момент, не содержит ссылок на него, либо цепочка ссылок, которая могла бы связать объект с некоторой сущностью приложения, обрывается);

Освобождать память от мусора.

Существует два подхода к обнаружению мусора:

Reference counting;

Tracing

Reference counting (подсчёт ссылок). Суть этого подхода состоит в том, что каждый объект имеет счетчик. Счетчик хранит информацию о том, сколько ссылок указывает на объект. Когда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором. Главным минусом такого подхода является сложность обеспечения точности счетчика. Также при таком подходе сложно выявлять циклические зависимости (когда два объекта указывают друг на друга, но ни один живой объект на них не ссылается), что приводит к утечкам памяти.

Главная идея подхода Tracing (трассировка) состоит в утверждении, что живыми могут считаться только те объекты, до которых мы можем добраться из корневых точек (GC Root) и те объекты, которые доступны с живого объекта. Всё остальное - мусор.

Существует 4 типа корневых точки:

Локальные переменные и параметры методов;

Потоки;

Статические переменные;

Ссылки из JNI.

Самое простое java приложение будет иметь корневые точки:

Локальные переменные внутри main() метода и параметры main() метода;

Поток который выполняет main();

Статические переменные класса, внутри которого находится main() метод.

Таким образом, если мы представим все объекты и ссылки между ними как дерево, то нам нужно будет пройти с корневых узлов (точек) по всем рёбрам. При этом узлы, до которых мы сможем добраться - не мусор, все остальные - мусор. При таком подходе циклические зависимости легко выявляются. HotSpot VM использует именно такой подход.

Для очистки памяти от мусора существуют два основных метода:

Copying collectors

Mark-and-sweep

При copying collectors подходе память делится на две части «from-space» и «to-space», при этом сам принцип работы такой:

Объекты создаются в «from-space»;

Когда «from-space» заполняется, приложение приостанавливается;

Запускается сборщик мусора. Находятся живые объекты в «from-space» и копируются в «to-space»;

Когда все объекты скопированы «from-space» полностью очищается;

«to-space» и «from-space» меняются местами.

Главный плюс такого подхода в том, что объекты плотно забивают память. Минусы подхода:

Приложение должно быть остановлено на время, необходимое для полного прохождения цикла сборки мусора;

В худшем случае (когда все объекты живые) «form-space» и «to-space» будут обязаны быть одинакового размера.

Алгоритм работы mark-and-sweep можно описать так:

Объекты создаются в памяти;

В момент, когда нужно запустить сборщик мусора приложение приостанавливается;

Сборщик проходится по дереву объектов, помечая живые объекты;

Сборщик проходится по всей памяти, находя все не отмеченные куски памяти и сохраняя их в «free list»;

Когда новые объекты начинают создаваться они создаются в памяти доступной во «free list».

Минусы этого способа:

Приложение не работает пока происходит сборка мусора;

Время остановки напрямую зависит от размеров памяти и количества объектов;

Если не использовать «compacting» память будет использоваться не эффективно.

Сборщики мусора HotSpot VM используют комбинированный подход Generational Garbage Collection, который позволяет использовать разные алгоритмы для разных этапов сборки мусора. Этот подход опирается на том, что:

большинство создаваемых объектов быстро становятся мусором;

существует мало связей между объектами, которые были созданы в прошлом и только что созданными объектами.

**Как работает сборщик мусора?**

Механизм сборки мусора - это процесс освобождения места в куче, для возможности добавления новых объектов.

Объекты создаются посредством оператора new, тем самым присваивая объекту ссылку. Для окончания работы с объектом достаточно просто перестать на него ссылаться, например, присвоив переменной ссылку на другой объект или значение null; прекратить выполнение метода, чтобы его локальные переменные завершили свое существование естественным образом. Объекты, ссылки на которые отсутствуют, принято называть мусором (garbage), который будет удален.

Виртуальная машина Java, применяя механизм сборки мусора, гарантирует, что любой объект, обладающий ссылками, остается в памяти — все объекты, которые недостижимы из исполняемого кода, ввиду отсутствия ссылок на них, удаляются с высвобождением отведенной для них памяти. Точнее говоря, объект не попадает в сферу действия процесса сборки мусора, если он достижим посредством цепочки ссылок, начиная с корневой (GC Root) ссылки, т.е. ссылки, непосредственно существующей в выполняемом коде.

Память освобождается сборщиком мусора по его собственному «усмотрению». Программа может успешно завершить работу, не исчерпав ресурсов свободной памяти или даже не приблизившись к этой черте и поэтому ей так и не потребуются «услуги» сборщика мусора.

Мусор собирается системой автоматически, без вмешательства пользователя или программиста, но это не значит, что этот процесс не требует внимания вовсе. Необходимость создания и удаления большого количества объектов существенным образом сказывается на производительности приложений и, если быстродействие программы является важным фактором, следует тщательно обдумывать решения, связанные с созданием объектов, — это, в свою очередь, уменьшит и объем мусора, подлежащего утилизации.

**Какие разновидности сборщиков мусора реализованы в виртуальной машине HotSpot?**

Java HotSpot VM предоставляет разработчикам на выбор четыре различных сборщика мусора:

Serial (последовательный) — самый простой вариант для приложений с небольшим объемом данных и не требовательных к задержкам. На данный момент используется сравнительно редко, но на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию. Использование Serial GC включается опцией -XX:+UseSerialGC.

Parallel (параллельный) — наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности. Параллельный сборщик включается опцией -XX:+UseParallelGC.

Concurrent Mark Sweep (CMS) — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти. Использование CMS GC включается опцией -XX:+UseConcMarkSweepGC.

Garbage-First (G1) — создан для замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных. G1 включается опцией Java -XX:+UseG1GC.

**Опишите алгоритм работы какого-нибудь сборщика мусора, реализованного в виртуальной машине HotSpot.**

Serial Garbage Collector (Последовательный сборщик мусора) был одним из первых сборщиков мусора в HotSpot VM. Во время работы этого сборщика приложения приостанавливается и продолжает работать только после прекращения сборки мусора.

Память приложения делится на три пространства:

Young generation. Объекты создаются именно в этом участке памяти.

Old generation. В этот участок памяти перемещаются объекты, которые переживают «minor garbage collection».

Permanent generation. Тут хранятся метаданные об объектах, Class data sharing (CDS), пул строк (String pool). Permanent область делится на две: только для чтения и для чтения-записи. Очевидно, что в этом случае область только для чтения не чистится сборщиком мусора никогда.

Область памяти Young generation состоит из трёх областей: Eden и двух меньших по размеру Survivor spaces - To space и From space. Большинство объектов создаются в области Eden, за исключением очень больших объектов, которые не могут быть размещены в ней и поэтому сразу размещаются в Old generation. В Survivor spaces перемещаются объекты, которые пережили по крайней мере одну сборку мусора, но ещё не достигли порога «старости» (tenuring threshold), чтобы быть перемещенными в Old generation.

Когда Young generation заполняется, то в этой области запускается процесс лёгкой сборки (minor collection), в отличие от процесса сборки, проводимого над всей кучей (full collection). Он происходит следующим образом: в начале работы одно из Survivor spaces - To space, является пустым, а другое - From space, содержит объекты, пережившие предыдущие сборки. Сборщик мусора ищет живые объекты в Eden и копирует их в To space, а затем копирует туда же и живые «молодые» (то есть не пережившие еще заданное число сборок мусора) объекты из From space. Старые объекты из From space перемещаются в Old generation. После лёгкой сборки From space и To space меняются ролями, область Eden становится пустой, а число объектов в Old generation увеличивается.

Если в процессе копирования живых объектов To space переполняется, то оставшиеся живые объекты из Eden и From space, которым не хватило места в To space, будут перемещены в Old generation, независимо от того, сколько сборок мусора они пережили.

Поскольку при использовании этого алгоритма сборщик мусора просто копирует все живые объекты из одной области памяти в другую, то такой сборщик мусора называется copying (копирующий). Очевидно, что для работы копирующего сборщика мусора у приложения всегда должна быть свободная область памяти, в которую будут копироваться живые объекты, и такой алгоритм может применяться для областей памяти сравнительно небольших по отношению к общему размеру памяти приложения. Young generation как раз удовлетворяет этому условию (по умолчанию на машинах клиентского типа эта область занимает около 10% кучи (значение может варьироваться в зависимости от платформы)).

Однако, для сборки мусора в Old generation, занимающем большую часть всей памяти, используется другой алгоритм.

В Old generation сборка мусора происходит с использованием алгоритма mark-sweep-compact, который состоит из трёх фаз. В фазе Mark (пометка) сборщик мусора помечает все живые объекты, затем, в фазе Sweep (очистка) все не помеченные объекты удаляются, а в фазе Сompact (уплотнение) все живые объекты перемещаются в начало Old generation, в результате чего свободная память после очистки представляет собой непрерывную область. Фаза уплотнения выполняется для того, чтобы избежать фрагментации и упростить процесс выделения памяти в Old generation.

Когда свободная память представляет собой непрерывную область, то для выделения памяти под создаваемый объект можно использовать очень быстрый (около десятка машинных инструкций) алгоритм bump-the-pointer: адрес начала свободной памяти хранится в специальном указателе, и когда поступает запрос на создание нового объекта, код проверяет, что для нового объекта достаточно места, и, если это так, то просто увеличивает указатель на размер объекта.

Последовательный сборщик мусора отлично подходит для большинства приложений, использующих до 200 мегабайт кучи, работающих на машинах клиентского типа и не предъявляющих жёстких требований к величине пауз, затрачиваемых на сборку мусора. В то же время модель «stop-the-world» может вызвать длительные паузы в работе приложения при использовании больших объёмов памяти. Кроме того, последовательный алгоритм работы не позволяет оптимально использовать вычислительные ресурсы компьютера, и последовательный сборщик мусора может стать узким местом при работе приложения на многопроцессорных машинах.

**Что такое «пул строк»?**

Пул строк – это набор строк, хранящийся в Heap.

Пул строк возможен благодаря неизменяемости строк в Java и реализации идеи интернирования строк;

Пул строк помогает экономить память, но по этой же причине создание строки занимает больше времени;

Когда для создания строки используются ", то сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка на неё;

При использовании оператора new создаётся новый объект String. Затем при помощи метода intern() эту строку можно поместить в пул или же получить из пула ссылку на другой объект String с таким же значением;

Пул строк является примером паттерна «Приспособленец» (Flyweight).

**Что такое finalize()? Зачем он нужен?**

Через вызов метода finalize() (который наследуется от Java.lang.Object) JVM реализуется функциональность аналогичная функциональности деструкторов в С++, используемых для очистки памяти перед возвращением управления операционной системе. Данный метод вызывается при уничтожении объекта сборщиком мусора (garbage collector) и переопределяя finalize() можно запрограммировать действия необходимые для корректного удаления экземпляра класса - например, закрытие сетевых соединений, соединений с базой данных, снятие блокировок на файлы и т.д.

После выполнения этого метода объект должен быть повторно собран сборщиком мусора (и это считается серьезной проблемой метода finalize() т.к. он мешает сборщику мусора освобождать память). Вызов этого метода не гарантируется, т.к. приложение может быть завершено до того, как будет запущена сборка мусора.

Объект не обязательно будет доступен для сборки сразу же - метод finalize() может сохранить куда-нибудь ссылку на объект. Подобная ситуация называется «возрождением» объекта и считается антипаттерном. Главная проблема такого трюка - в том, что «возродить» объект можно только 1 раз.

Пример:

public class MainClass {

public static void main(String args[]) {

TestClass a = new TestClass();

a.a();

a = null;

a = new TestClass();

a.a();

System.out.println("!!! done");

}

}

public class TestClass {

public void a() {

System.out.println("!!! a() called");

}

@Override

protected void finalize() throws Throwable {

System.out.println("!!! finalize() called");

super.finalize();

}

}

Так как в данном случае сборщик мусора может и не быть вызван (в силу простоты приложения), то результат выполнения программы с большой вероятностью будет следующий:

!!! a() called

!!! a() called

!!! done

Теперь несколько усложним программу, добавив принудительный вызов Garbage Collector:

public class MainClass {

public static void main(String args[]) {

TestClass a = new TestClass();

a.a();

a = null;

System.gc(); // Принудительно зовём сборщик мусора

a = new TestClass();

a.a();

System.out.println("!!! done");

}

}

Как и было сказано ранее, Garbage Collector может в разное время отработать, поэтому результат выполнения может разниться от запуска к запуску: Вариант а:

!!! a() called

!!! a() called

!!! done

!!! finalize() called

Вариант б:

!!! a() called

!!! a() called

!!! finalize() called

!!! done

**Что произойдет со сборщиком мусора, если выполнение метода finalize() требует ощутимо много времени, или в процессе выполнения будет выброшено исключение?**

Непосредственно вызов finalize() происходит в отдельном потоке Finalizer (java.lang.ref.Finalizer.FinalizerThread), который создаётся при запуске виртуальной машины (в статической секции при загрузке класса Finalizer). Методы finalize() вызываются последовательно в том порядке, в котором были добавлены в список сборщиком мусора. Соответственно, если какой-то finalize() зависнет, он подвесит поток Finalizer, но не сборщик мусора. Это в частности означает, что объекты, не имеющие метода finalize(), будут исправно удаляться, а вот имеющие будут добавляться в очередь, пока поток Finalizer не освободится, не завершится приложение или не кончится память.

То же самое применимо и выброшенным в процессе finalize() исключениям: метод runFinalizer() у потока Finalizer игнорирует все исключения выброшенные в момент выполнения finalize(). Таким образом возникновение исключительной ситуации никак не скажется на работоспособности сборщика мусора.

**Чем отличаются final, finally и finalize()?**

Модификатор final:

Класс не может иметь наследников;

Метод не может быть переопределен в классах наследниках;

Поле не может изменить свое значение после инициализации;

Локальные переменные не могут быть изменены после присвоения им значения;

Параметры методов не могут изменять своё значение внутри метода.

Оператор finally гарантирует, что определенный в нём участок кода будет выполнен независимо от того, какие исключения были возбуждены и перехвачены в блоке try-catch.

Метод finalize() вызывается перед тем как сборщик мусора будет проводить удаление объекта.

Пример:

public class MainClass {

public static void main(String args[]) {

TestClass a = new TestClass();

System.out.println("result of a.a() is " + a.a());

a = null;

System.gc(); // Принудительно зовём сборщик мусора

a = new TestClass();

System.out.println("result of a.a() is " + a.a());

System.out.println("!!! done");

}

}

public class TestClass {

public int a() {

try {

System.out.println("!!! a() called");

throw new Exception("");

} catch (Exception e) {

System.out.println("!!! Exception in a()");

return 2;

} finally {

System.out.println("!!! finally in a() ");

}

}

@Override

protected void finalize() throws Throwable {

System.out.println("!!! finalize() called");

super.finalize();

}

}

Результат выполнения:

!!! a() called

!!! Exception in a()

!!! finally in a()

result of a.a() is 2

!!! a() called

!!! Exception in a()

!!! finally in a()

!!! finalize() called

result of a.a() is 2

!!! done

**Расскажите про приведение типов. Что такое понижение и повышение типа?**

Java является строго типизированным языком программирования, а это означает, то что каждое выражение и каждая переменная имеет строго определенный тип уже на момент компиляции. Однако определен механизм приведения типов (casting) - способ преобразования значения переменной одного типа в значение другого типа.

В Java существуют несколько разновидностей приведения:

Тождественное (identity). Преобразование выражения любого типа к точно такому же типу всегда допустимо и происходит автоматически.

Расширение (повышение, upcasting) примитивного типа (widening primitive). Означает, что осуществляется переход от менее емкого типа к более ёмкому. Например, от типа byte (длина 1 байт) к типу int (длина 4 байта). Такие преобразование безопасны в том смысле, что новый тип всегда гарантировано вмещает в себя все данные, которые хранились в старом типе и таким образом не происходит потери данных. Этот тип приведения всегда допустим и происходит автоматически.

Сужение (понижение, downcasting) примитивного типа (narrowing primitive). Означает, что переход осуществляется от более емкого типа к менее емкому. При таком преобразовании есть риск потерять данные. Например, если число типа int было больше 127, то при приведении его к byte значения битов старше восьмого будут потеряны. В Java такое преобразование должно совершаться явным образом, при этом все старшие биты, не умещающиеся в новом типе, просто отбрасываются - никакого округления или других действий для получения более корректного результата не производится.

Расширение объектного типа (widening reference). Означает неявное восходящее приведение типов или переход от более конкретного типа к менее конкретному, т.е. переход от потомка к предку. Разрешено всегда и происходит автоматически.

Сужение объектного типа (narrowing reference). Означает нисходящее приведение, то есть приведение от предка к потомку (подтипу). Возможно только если исходная переменная является подтипом приводимого типа. При несоответствии типов в момент выполнения выбрасывается исключение ClassCastException. Требует явного указания типа.

Преобразование к строке (to String). Любой тип может быть приведен к строке, т.е. к экземпляру класса String.

Запрещенные преобразования (forbidden). Не все приведения между произвольными типами допустимы. Например, к запрещенным преобразованиям относятся приведения от любого ссылочного типа к примитивному и наоборот (кроме преобразования к строке). Кроме того, невозможно привести друг к другу классы, находящиеся на разных ветвях дерева наследования и т.п.

При приведении ссылочных типов с самим объектом ничего не происходит, - меняется лишь тип ссылки, через которую происходит обращение к объекту.

Для проверки возможности приведения нужно воспользоваться оператором instanceof:

Parent parent = new Child();

if (parent instanceof Child) {

Child child = (Child) parent;

}

**Когда в приложении может быть выброшено исключение ClassCastException?**

ClassCastException (потомок RuntimeException) - исключение, которое будет выброшено при ошибке приведения типа.

**Что такое литералы?**

Литералы — это явно заданные значения в коде программы — константы определенного типа, которые находятся в коде в момент запуска.

class Test {

int a = 0b1101010110;

public static void main(String[] args) {

System.out.println("Hello world!");

}

}

В этом классе “Hello world!” — литерал.

Переменная a - тоже литерал.

Литералы бывают разных типов, которые определяются их назначением и способом написания.

**Что такое autoboxing («автоупаковка») в Java и каковы правила упаковки примитивных типов в классы-обертки?**

Автоупаковка - это механизм неявной инициализации объектов классов-оберток (Byte, Short, Integer, Long, Float, Double, Character, Boolean) значениями соответствующих им исходных примитивных типов (byte, short, int...), без явного использования конструктора класса.

Автоупаковка происходит при прямом присваивании примитива классу-обертке (с помощью оператора =), либо при передаче примитива в параметры метода (типа класса-обертки).

Автоупаковке в классы-обертки могут быть подвергнуты как переменные примитивных типов, так и константы времени компиляции (литералы и final-примитивы). При этом литералы должны быть синтаксически корректными для инициализации переменной исходного примитивного типа.

Автоупаковка переменных примитивных типов требует точного соответствия типа исходного примитива типу класса-обертки. Например, попытка упаковать переменную типа byte в Short, без предварительного явного приведения byte в short вызовет ошибку компиляции.

Автоупаковка констант примитивных типов допускает более широкие границы соответствия. В этом случае компилятор способен предварительно осуществлять неявное расширение/сужение типа примитивов:

неявное расширение/сужение исходного типа примитива до типа примитива, соответствующего классу-обертке (для преобразования int в Byte, сначала компилятор самостоятельно неявно сужает int к byte)

автоупаковку примитива в соответствующий класс-обертку. Однако, в этом случае существуют два дополнительных ограничения: a) присвоение примитива обертке может производится только оператором = (нельзя передать такой примитив в параметры метода без явного приведения типов) b) тип левого операнда не должен быть старше чем Character, тип правого не должен старше, чем int: допустимо расширение/сужение byte в/из short, byte в/из char, short в/из char и только сужение byte из int, short из int, char из int. Все остальные варианты требуют явного приведения типов).

Дополнительной особенностью целочисленных классов-оберток, созданных автоупаковкой констант в диапазоне -128 ... +127 является то, что они кэшируются JVM. Поэтому такие обертки с одинаковыми значениями будут являться ссылками на один объект.

**Какие есть особенности класса String**?

Это неизменяемый (immutable) и финализированный тип данных;

Все объекты класса String JVM хранит в пуле строк;

Объект класса String можно получить, используя двойные кавычки;

Можно использовать оператор + для конкатенации строк;

Начиная с Java 7 строки можно использовать в конструкции switch.

**Почему String неизменяемый и финализированный класс?**

Есть несколько преимуществ в неизменности строк:

Пул строк возможен только потому, что строка неизменяемая, таким образом виртуальная машина сохраняет больше свободного места в Heap, поскольку разные строковые переменные указывают на одну и ту же переменную в пуле. Если бы строка была изменяемой, то интернирование строк не было бы возможным, потому что изменение значения одной переменной отразилось бы также и на остальных переменных, ссылающихся на эту строку.

Если строка будет изменяемой, тогда это станет серьезной угрозой безопасности приложения. Например, имя пользователя базы данных и пароль передаются строкой для получения соединения с базой данных и в программировании сокетов реквизиты хоста и порта передаются строкой. Так как строка неизменяемая, её значение не может быть изменено, в противном случае злоумышленник может изменить значение ссылки и вызвать проблемы в безопасности приложения.

Неизменяемость позволяет избежать синхронизации: строки безопасны для многопоточности и один экземпляр строки может быть совместно использован различными потоками.

Строки используются classloader и неизменность обеспечивает правильность загрузки класса.

Поскольку строка неизменяемая, её hashCode() кэшируется в момент создания и нет необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в HashMap т.к. его обработка происходит быстрее.

**Почему char[] предпочтительнее String для хранения пароля?**

С момента создания строка остаётся в пуле, до тех пор, пока не будет удалена сборщиком мусора. Поэтому, даже после окончания использования пароля, он некоторое время продолжает оставаться доступным в памяти и способа избежать этого не существует. Это представляет определённый риск для безопасности, поскольку кто-либо, имеющий доступ к памяти сможет найти пароль в виде текста. В случае использования массива символов для хранения пароля имеется возможность очистить его сразу по окончанию работы с паролем, позволяя избежать риска безопасности, свойственного строке.

**Почему строка является популярным ключом в HashMap в Java?**

Поскольку строки неизменяемы, их хэш код вычисляется и кэшируется в момент создания, не требуя повторного пересчета при дальнейшем использовании. Поэтому в качестве ключа HashMap они будут обрабатываться быстрее.

**Что делает метод intern() в классе String?**

Метод intern() используется для сохранения строки в пуле строк или получения ссылки, если такая строка уже находится в пуле.

**Можно ли использовать строки в конструкции switch?**

Да, начиная с Java 7 в операторе switch можно использовать строки, ранние версии Java не поддерживают этого. При этом:

участвующие строки чувствительны к регистру;

используется метод equals() для сравнения полученного значения со значениями case, поэтому во избежание NullPointerException стоит предусмотреть проверку на null.

согласно документации, Java 7 для строк в switch, компилятор Java формирует более эффективный байткод для строк в конструкции switch, чем для сцепленных условий if-else.

**Какая основная разница между String, StringBuffer, StringBuilder?**

Класс String является неизменяемым (immutable) - модифицировать объект такого класса нельзя, можно лишь заменить его созданием нового экземпляра.

Класс StringBuffer изменяемый - использовать StringBuffer следует тогда, когда необходимо часто модифицировать содержимое.

Класс StringBuilder был добавлен в Java 5 и он во всем идентичен классу StringBuffer за исключением того, что он не синхронизирован и поэтому его методы выполняются значительно быстрей.

**Что такое класс Object? Какие в нем есть методы?**

Object это базовый класс для всех остальных объектов в Java. Любой класс наследуется от Object и, соответственно, наследуют его методы:

public boolean equals(Object obj) – служит для сравнения объектов по значению;

int hashCode() – возвращает hash код для объекта;

String toString() – возвращает строковое представление объекта;

Class getClass() – возвращает класс объекта во время выполнения;

protected Object clone() – создает и возвращает копию объекта;

void notify() – возобновляет поток, ожидающий монитор;

void notifyAll() – возобновляет все потоки, ожидающие монитор;

void wait() – остановка вызвавшего метод потока до момента пока другой поток не вызовет метод notify() или notifyAll() для этого объекта;

void wait(long timeout) – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод notify() или notifyAll() для этого объекта;

void wait(long timeout, int nanos) – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод notify() или notifyAll() для этого объекта;

protected void finalize() – может вызываться сборщиком мусора в момент удаления объекта при сборке мусора.

**Дайте определение понятию «конструктор».**

Конструктор — это специальный метод, у которого отсутствует возвращаемый тип и который имеет то же имя, что и класс, в котором он используется. Конструктор вызывается при создании нового объекта класса и определяет действия необходимые для его инициализации.

**Что такое «конструктор по умолчанию»?**

Если у какого-либо класса не определить конструктор, то компилятор сгенерирует конструктор без аргументов - так называемый «конструктор по умолчанию».

public class ClassName() {}

Если у класса уже определен какой-либо конструктор, то конструктор по умолчанию создан не будет и, если он необходим, его нужно описывать явно.

**Чем отличаются конструктор по-умолчанию, конструктор копирования и конструктор с параметрами?**

У конструктора по умолчанию отсутствуют какие-либо аргументы. Конструктор копирования принимает в качестве аргумента уже существующий объект класса для последующего создания его клона. Конструктор с параметрами имеет в своей сигнатуре аргументы (обычно необходимые для инициализации полей класса).

**Где и как вы можете использовать приватный конструктор?**

Приватный (помеченный ключевым словом private, скрытый) конструктор может использоваться публичным статическим методом генерации объектов данного класса. Также доступ к нему разрешён вложенным классам и может использоваться для их нужд.

**Расскажите про классы-загрузчики и про динамическую загрузку классов.**

Основа работы с классами в Java — классы-загрузчики, обычные Java-объекты, предоставляющие интерфейс для поиска и создания объекта класса по его имени во время работы приложения.

В начале работы программы создается 3 основных загрузчика классов:

базовый загрузчик (bootstrap/primordial). Загружает основные системные и внутренние классы JDK (Core API - пакеты java.\* (rt.jar и i18n.jar) . Важно заметить, что базовый загрузчик является «Изначальным» или «Корневым» и частью JVM, вследствие чего его нельзя создать внутри кода программы.

загрузчик расширений (extention). Загружает различные пакеты расширений, которые располагаются в директории <JAVA\_HOME>/lib/ext или другой директории, описанной в системном параметре java.ext.dirs. Это позволяет обновлять и добавлять новые расширения без необходимости модифицировать настройки используемых приложений. Загрузчик расширений реализован классом sun.misc.Launcher$ExtClassLoader.

системный загрузчик (system/application). Загружает классы, пути к которым указаны в переменной окружения CLASSPATH или пути, которые указаны в командной строке запуска JVM после ключей -classpath или -cp. Системный загрузчик реализован классом sun.misc.Launcher$AppClassLoader.

Загрузчики классов являются иерархическими: каждый из них (кроме базового) имеет родительский загрузчик и в большинстве случаев, перед тем как попробовать загрузить класс самостоятельно, он посылает вначале запрос родительскому загрузчику загрузить указанный класс. Такое делегирование позволяет загружать классы тем загрузчиком, который находится ближе всего к базовому в иерархии делегирования. Как следствие поиск классов будет происходить в источниках в порядке их доверия: сначала в библиотеке Core API, потом в папке расширений, потом в локальных файлах CLASSPATH.

Процесс загрузки класса состоит из трех частей:

Loading – на этой фазе происходит поиск и физическая загрузка файла класса в определенном источнике (в зависимости от загрузчика). Этот процесс определяет базовое представление класса в памяти. На этом этапе такие понятия как «методы», «поля» и т.д. пока не известны.

Linking – процесс, который может быть разбит на 3 части:

Bytecode verification – проверка байт-кода на соответствие требованиям, определенным в спецификации JVM.

Class preparation – создание и инициализация необходимых структур, используемых для представления полей, методов, реализованных интерфейсов и т.п., определенных в загружаемом классе.

Resolving – загрузка набора классов, на которые ссылается загружаемый класс.

Initialization – вызов статических блоков инициализации и присваивание полям класса значений по умолчанию.

Динамическая загрузка классов в Java имеет ряд особенностей:

отложенная (lazy) загрузка и связывание классов. Загрузка классов производится только при необходимости, что позволяет экономить ресурсы и распределять нагрузку.

проверка корректности загружаемого кода (type safeness). Все действия связанные с контролем использования типов производятся только во время загрузки класса, позволяя избежать дополнительной нагрузки во время выполнения кода.

программируемая загрузка. Пользовательский загрузчик полностью контролирует процесс получения запрошенного класса — самому ли искать байт-код и создавать класс или делегировать создание другому загрузчику. Дополнительно существует возможность выставлять различные атрибуты безопасности для загружаемых классов, позволяя таким образом работать с кодом из ненадежных источников.

множественные пространства имен. Каждый загрузчик имеет своё пространство имён для создаваемых классов. Соответственно, классы, загруженные двумя различными загрузчиками на основе общего байт-кода, в системе будут различаться.

Существует несколько способов инициировать загрузку требуемого класса:

явный: вызов ClassLoader.loadClass() или Class.forName() (по умолчанию используется загрузчик, создавший текущий класс, но есть возможность и явного указания загрузчика);

неявный: когда для дальнейшей работы приложения требуется ранее не использованный класс, JVM инициирует его загрузку.

**Что такое Reflection?**

Рефлексия (Reflection) - это механизм получения данных о программе во время её выполнения (runtime). В Java Reflection осуществляется с помощью Java Reflection API, состоящего из классов пакетов java.lang и java.lang.reflect.

Возможности Java Reflection API:

Определение класса объекта;

Получение информации о модификаторах класса, полях, методах, конструкторах и суперклассах;

Определение интерфейсов, реализуемых классом;

Создание экземпляра класса;

Получение и установка значений полей объекта;

Вызов методов объекта;

Создание нового массива.

**Зачем нужен equals(). Чем он отличается от операции ==?**

Метод equals() - определяет отношение эквивалентности объектов.

При сравнении объектов с помощью == сравнение происходит лишь между ссылками. При сравнении по переопределённому разработчиком equals() - по внутреннему состоянию объектов.

**Какими свойствами обладает порождаемое equals() отношение эквивалентности?**

Рефлексивность: для любой ссылки на значение x, x.equals(x) вернет true;

Симметричность: для любых ссылок на значения x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) возвращает true.

Транзитивность: для любых ссылок на значения x, y и z, если x.equals(y) и y.equals(z) возвращают true, тогда и x.equals(z) вернёт true;

Непротиворечивость: для любых ссылок на значения х и у, если несколько раз вызвать х.equals(y), постоянно будет возвращаться значение true либо постоянно будет возвращаться значение false при условии, что никакая информация, используемая при сравнении объектов, не поменялась.

Для любой ненулевой ссылки на значение х выражение х.equals(null) должно возвращать false.

Правила переопределения метода Object.equals().

Использование оператора == для проверки, является ли аргумент ссылкой на указанный объект. Если является, возвращается true. Если сравниваемый объект == null, должно вернуться false.

Использование оператор instanceof и вызова метода getClass() для проверки, имеет ли аргумент правильный тип. Если не имеет, возвращается false.

Приведение аргумента к правильному типу. Поскольку эта операция следует за проверкой instanceof она гарантированно будет выполнена.

Обход всех значимых полей класса и проверка того, что значение поля в текущем объекте и значение того же поля в проверяемом на эквивалентность аргументе соответствуют друг другу. Если проверки для всех полей прошли успешно, возвращается результат true, в противном случае - false.

По окончанию переопределения метода equals() следует проверить: является ли порождаемое отношение эквивалентности рефлексивным, симметричным, транзитивным и непротиворечивым? Если ответ отрицательный, метод подлежит соответствующей правке.

**Если equals() переопределен, есть ли какие-либо другие методы, которые следует переопределить?**

Равные объекты должны возвращать одинаковые хэш коды. При переопределении equals() нужно обязательно переопределять и метод hashCode().

**Что будет, если переопределить equals() не переопределяя hashCode()? Какие могут возникнуть проблемы?**

Классы и методы, которые используют правила этого контракта могут работать некорректно. Так для HashMap это может привести к тому, что пара «ключ-значение», которая была в неё помещена при использовании нового экземпляра ключа не будет в ней найдена.

**Каким образом реализованы методы hashCode() и equals() в классе Object?**

Реализация метода Object.equals() сводится к проверке на равенство двух ссылок:

public boolean equals(Object obj) {

return (this == obj);

}

Реализация метода Object.hashCode() описана как native, т.е. определенной не с помощью Java кода и обычно возвращает адрес объекта в памяти:

public native int hashCode();

**Для чего нужен метод hashCode()?**

Метод hashCode() необходим для вычисления хэш кода переданного в качестве входного параметра объекта. В Java это целое число, в более широком смысле - битовая строка фиксированной длины, полученная из массива произвольной длины. Этот метод реализован таким образом, что для одного и того же входного объекта, хэш код всегда будет одинаковым. Следует понимать, что в Java множество возможных хэш кодов ограничено типом int, а множество объектов ничем не ограничено. Из-за этого, вполне возможна ситуация, что хэш коды разных объектов могут совпасть:

если хэш коды разные, то и объекты гарантированно разные;

если хэш коды равны, то объекты не обязательно равны(могут быть разные).

**Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode()?**

Общий совет: выбирать поля, которые с большой долью вероятности будут различаться. Для этого необходимо использовать уникальные, лучше всего примитивные поля, например, такие как id, uuid. При этом нужно следовать правилу, если поля задействованы при вычислении hashCode(), то они должны быть задействованы и при выполнении equals().

**Могут ли у разных объектов быть одинаковые hashCode()?**

Да, могут. Метод hashCode() не гарантирует уникальность возвращаемого значения. Ситуация, когда у разных объектов одинаковые хэш коды называется коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хэш кода.

**Если у класса Point{int x, y;} реализовать метод equals(Object that) {(return this.x == that.x && this.y == that.y)}, но сделать хэш код в виде int hashCode() {return x;}, то будут ли корректно такие точки помещаться и извлекаться из HashSet?**

HashSet использует HashMap для хранения элементов. При добавлении элемента в HashMap вычисляется хэш код, по которому определяется позиция в массиве, куда будет вставлен новый элемент. У всех экземпляров класса Point хэш код будет одинаковым для всех объектов с одинаковым x, что приведёт к вырождению хэш таблицы в список.

При возникновении коллизии в HashMap осуществляется проверка на наличие элемента в списке: e.hash == hash && ((k = e.key) == key || key.equals(k)). Если элемент найден, то его значение перезаписывается. В нашем случае для разных объектов метод equals() будет возвращать false. Соответственно новый элемент будет успешно добавлен в HashSet. Извлечение элемента также будет осуществляться успешно. Но производительность такого кода будет невысокой и преимущества хэш таблиц использоваться не будут.

**Могут ли у разных объектов (ref0 != ref1) быть ref0.equals(ref1) == true?**

Да, могут. Для этого в классе этих объектов должен быть переопределен метод equals().

Если используется метод Object.equals(), то для двух ссылок x и y метод вернет true тогда и только тогда, когда обе ссылки указывают на один и тот же объект (т.е. x == y возвращает true).

**Могут ли у разных ссылок на один объект (ref0 == ref1) быть ref0.equals(ref1) == false?**

В общем случае - могут, если метод equals() реализован некорректно и не выполняет свойство рефлексивности: для любых ненулевых ссылок x метод x.equals(x) должен возвращать true.

**Можно ли так реализовать метод equals(Object that) {return this.hashCode() == that.hashCode()}?**

Строго говоря нельзя, поскольку метод hashCode() не гарантирует уникальность значения для каждого объекта. Однако для сравнения экземпляров класса Object такой код допустим, т.к. метод hashCode() в классе Object возвращает уникальные значения для разных объектов (его вычисление основано на использовании адреса объекта в памяти).

**В equals() требуется проверять, что аргумент equals(Object that) такого же типа что и сам объект. В чем разница между this.getClass() == that.getClass() и that instanceof MyClass?**

Оператор instanceof сравнивает объект и указанный тип. Его можно использовать для проверки является ли данный объект экземпляром некоторого класса, либо экземпляром его дочернего класса, либо экземпляром класса, который реализует указанный интерфейс.

this.getClass() == that.getClass() проверяет два класса на идентичность, поэтому для корректной реализации контракта метода equals() необходимо использовать точное сравнение с помощью метода getClass().

**Можно ли реализовать метод equals() класса MyClass вот так: class MyClass {public boolean equals(MyClass that) {return this == that;}}?**

Реализовать можно, но данный метод не переопределяет метод equals() класса Object, а перегружает его.

**Есть класс Point{int x, y;}. Почему хэш код в виде 31 \* x + y предпочтительнее чем x + y?**

Множитель создает зависимость значения хэш кода от очередности обработки полей, что в итоге порождает лучшую хэш функцию.

**Расскажите про клонирование объектов.**

Использование оператора присваивания не создает нового объекта, а лишь копирует ссылку на объект. Таким образом, две ссылки указывают на одну и ту же область памяти, на один и тот же объект. Для создания нового объекта с таким же состоянием используется клонирование объекта.

Класс Object содержит protected метод clone(), осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод clone() как public для обеспечения возможности его вызова. В переопределенном методе следует вызвать базовую версию метода super.clone(), которая и выполняет собственно клонирование.

Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс Cloneable. Интерфейс Cloneable не содержит методов относится к маркерным интерфейсам, а его реализация гарантирует, что метод clone() класса Object возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение CloneNotSupportedException. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора.

Это решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их обёрток или неизменяемых (immutable) объектных типов. Если же поле клонируемого типа является изменяемым ссылочным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и сам объект поля класса.

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс Cloneable и переопределяет метод clone(). Так как, если это будет иначе вызов метода невозможен из-за его недоступности. Отсюда следует, что если класс имеет суперкласс, то для реализации механизма клонирования текущего класса-потомка необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений final для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

Помимо встроенного механизма клонирования в Java для клонирования объекта можно использовать:

Специализированный конструктор копирования - в классе описывается конструктор, который принимает объект этого же класса и инициализирует поля создаваемого объекта значениями полей переданного.

Фабричный метод - (Factory method), который представляет собой статический метод, возвращающий экземпляр своего класса.

Механизм сериализации - сохранение и последующее восстановление объекта в/из потока байтов.

**В чем отличие между поверхностным и глубоким клонированием?**

Поверхностное копирование копирует настолько малую часть информации об объекте, насколько это возможно. По умолчанию, клонирование в Java является поверхностным, т.е. класс Object не знает о структуре класса, которого он копирует. Клонирование такого типа осуществляется JVM по следующим правилам:

Если класс имеет только члены примитивных типов, то будет создана совершенно новая копия объекта и возвращена ссылка на этот объект.

Если класс помимо членов примитивных типов содержит члены ссылочных типов, то тогда копируются ссылки на объекты этих классов. Следовательно, оба объекта будут иметь одинаковые ссылки.

Глубокое копирование дублирует абсолютно всю информацию объекта:

Нет необходимости копировать отдельно примитивные данные;

Все члены ссылочного типа в оригинальном классе должны поддерживать клонирование. Для каждого такого члена при переопределении метода clone() должен вызываться super.clone();

Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.

**Какой способ клонирования предпочтительней?**

Наиболее безопасным и, следовательно, предпочтительным способом клонирования является использование специализированного конструктора копирования:

Отсутствие ошибок наследования (не нужно беспокоиться, что у наследников появятся новые поля, которые не будут склонированы через метод clone());

Поля для клонирования указываются явно;

Возможность клонировать даже final поля.

**Почему метод clone() объявлен в классе Object, а не в интерфейсе Cloneable?**

Метод clone() объявлен в классе Object с указанием модификатора native, чтобы обеспечить доступ к стандартному механизму поверхностного копирования объектов. Одновременно он объявлен и как protected, чтобы нельзя было вызвать этот метод у не переопределивших его объектов. Непосредственно интерфейс Cloneable является маркерным (не содержит объявлений методов) и нужен только для обозначения самого факта, что данный объект готов к тому, чтобы быть клонированным. Вызов переопределённого метода clone() у не Cloneable объекта вызовет выбрасывание CloneNotSupportedException.

**Опишите иерархию исключений.**

Исключения делятся на несколько классов, но все они имеют общего предка — класс Throwable, потомками которого являются классы Exception и Error.

Ошибки (Errors) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память доступная виртуальной машине.

Исключения (Exceptions) являются результатом проблем в программе, которые в принципе решаемы, предсказуемы и последствия которых возможно устранить внутри программы. Например, произошло деление целого числа на ноль.

**Что такое checked и unchecked exception?**

В Java все исключения делятся на два типа:

checked (контролируемые/проверяемые исключения) должны обрабатываться блоком catch или описываться в заголовке метода (например, throws IOException). Наличие такого обработчика/модификатора в заголовке метода проверяется на этапе компиляции;

unchecked (неконтролируемые/непроверяемые исключения), к которым относятся ошибки Error (например, OutOfMemoryError), обрабатывать которые не рекомендуется и исключения времени выполнения, представленные классом RuntimeException и его наследниками (например, NullPointerException), которые могут не обрабатываться блоком catch и не быть описанными в заголовке метода.

**Какой оператор позволяет принудительно выбросить исключение?**

Это оператор throw:

throw new Exception();

**О чем говорит ключевое слово throws?**

Модификатор throws прописывается в заголовке метода и указывает на то, что метод потенциально может выбросить исключение с указанным типом.

Как написать собственное («пользовательское») исключение?

Необходимо унаследоваться от базового класса требуемого типа исключений (например, от Exception или RuntimeException).

class CustomException extends Exception {

public CustomException() {

super();

}

public CustomException(final String string) {

super(string + " is invalid");

}

public CustomException(final Throwable cause) {

super(cause);

}

}

**Какие существуют unchecked exception?**

Наиболее часто встречающиеся: ArithmeticException, ClassCastException, ConcurrentModificationException, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, NoSuchElementException, NullPointerException, UnsupportedOperationException.

**Что представляет из себя ошибки класса Error?**

Ошибки класса Error представляют собой наиболее серьёзные проблемы уровня JVM. Например, исключения такого рода возникают, если закончилась память доступная виртуальной машине. Обрабатывать такие ошибки не запрещается, но делать этого не рекомендуется.

**Что вы знаете о OutOfMemoryError?**

OutOfMemoryError выбрасывается, когда виртуальная машина Java не может создать (разместить) объект из-за нехватки памяти, а сборщик мусора не может высвободить достаточное её количество.

Область памяти, занимаемая java процессом, состоит из нескольких частей. Тип OutOfMemoryError зависит от того, в какой из них не хватило места:

java.lang.OutOfMemoryError: Java heap space: Не хватает места в куче, а именно, в области памяти в которую помещаются объекты, создаваемые в приложении программно. Обычно проблема кроется в утечке памяти. Размер задается параметрами -Xms и -Xmx.

java.lang.OutOfMemoryError: PermGen space: (до версии Java 8) Данная ошибка возникает при нехватке места в Permanent области, размер которой задается параметрами -XX:PermSize и -XX:MaxPermSize.

java.lang.OutOfMemoryError: GC overhead limit exceeded: Данная ошибка может возникнуть как при переполнении первой, так и второй областей. Связана она с тем, что памяти осталось мало и сборщик мусора постоянно работает, пытаясь высвободить немного места. Данную ошибку можно отключить с помощью параметра -XX:-UseGCOverheadLimit.

java.lang.OutOfMemoryError: unable to create new native thread: Выбрасывается, когда нет возможности создавать новые потоки.

**Опишите работу блока try-catch-finally.**

try — данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке. catch — ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений в случае их возникновения. finally — ключевое слово для отметки начала блока кода, который является дополнительным. Этот блок помещается после последнего блока catch. Управление передаётся в блок finally в любом случае, было выброшено исключение или нет.

Общий вид конструкции для обработки исключительной ситуации выглядит следующим образом:

try {

//код, который потенциально может привести к исключительной ситуации

}

catch(SomeException e ) { //в скобках указывается класс конкретной ожидаемой ошибки

//код обработки исключительной ситуации

}

finally {

//необязательный блок, код которого выполняется в любом случае

}

**Что такое механизм try-with-resources?**

Данная конструкция, которая появилась в Java 7, позволяет использовать блок try-catch не заботясь о закрытии ресурсов, используемых в данном сегменте кода. Ресурсы объявляются в скобках сразу после try, а компилятор уже сам неявно создаёт секцию finally, в которой и происходит освобождение занятых в блоке ресурсов. Под ресурсами подразумеваются сущности, реализующие интерфейс java.lang.Autocloseable.

Общий вид конструкции:

try(/\*объявление ресурсов\*/) {

//...

} catch(Exception ex) {

//...

} finally {

//...

}

Стоит заметить, что блоки catch и явный finally выполняются уже после того, как закрываются ресурсы в неявном finally.

**Возможно ли использование блока try-finally (без catch)?**

Такая запись допустима, но смысла в такой записи не так много, всё же лучше иметь блок catch, в котором будет обрабатываться необходимое исключение.

**Может ли один блок catch отлавливать сразу несколько исключений?**

В Java 7 стала доступна новая языковая конструкция, с помощью которой можно перехватывать несколько исключений одним блоком catch:

try {

//...

} catch(IOException | SQLException ex) {

//...

}

**Всегда ли исполняется блок finally?**

Код в блоке finally будет выполнен всегда, независимо от того, выброшено исключение или нет.

**Существуют ли ситуации, когда блок finally не будет выполнен?**

Например, когда JVM «умирает» - в такой ситуации finally недостижим и не будет выполнен, так как происходит принудительный системный выход из программы:

try {

System.exit(0);

} catch(Exception e) {

e.printStackTrace();

} finally { }

**Может ли метод main() выбросить исключение во вне и если да, то где будет происходить обработка данного исключения?**

Может и оно будет передано в виртуальную машину Java (JVM).

Предположим, есть метод, который может выбросить IOException и FileNotFoundException в какой последовательности должны идти блоки catch? Сколько блоков catch будет выполнено?

Общее правило: обрабатывать исключения нужно от «младшего» к старшему. Т.е. нельзя поставить в первый блок catch(Exception ex) {}, иначе все дальнейшие блоки catch() уже ничего не смогут обработать, т.к. любое исключение будет соответствовать обработчику catch(Exception ex).

Таким образом, исходя из факта, что FileNotFoundException extends IOException сначала нужно обработать FileNotFoundException, а затем уже IOException:

void method() {

try {

//...

} catch (FileNotFoundException ex) {

//...

} catch (IOException ex) {

//...

}

}

**Что такое generics?**

Generics - это технический термин, обозначающий набор свойств языка позволяющих определять и использовать обобщенные типы и методы. Обобщенные типы или методы отличаются от обычных тем, что имеют типизированные параметры.

Примером использования обобщенных типов может служить Java Collection Framework. Так, класс LinkedList<E> - типичный обобщенный тип. Он содержит параметр E, который представляет тип элементов, которые будут храниться в коллекции. Создание объектов обобщенных типов происходит посредством замены параметризированных типов реальными типами данных. Вместо того, чтобы просто использовать LinkedList, ничего не говоря о типе элемента в списке, предлагается использовать точное указание типа LinkedList<String>, LinkedList<Integer> и т.п.

**Что такое «интернационализация», «локализация»?**

Интернационализация (internationalization) - способ создания приложений, при котором их можно легко адаптировать для разных аудиторий, говорящих на разных языках.

Локализация (localization) - адаптация интерфейса приложения под несколько языков. Добавление нового языка может внести определенные сложности в локализацию интерфейса.

**Что такое «коллекция»?**

«Коллекция» - это структура данных, набор каких-либо объектов. Данными (объектами в наборе) могут быть числа, строки, объекты пользовательских классов и т.п.

**Назовите основные интерфейсы JCF и их реализации.**

На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие во фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» соответственно.

Интерфейс Collection расширяют интерфейсы:

List (список) представляет собой коллекцию, в которой допустимы дублирующие значения. Реализации:

ArrayList - инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

LinkedList (двунаправленный связный список) - состоит из узлов, каждый из которых содержит как собственно данные, так и две ссылки на следующий и предыдущий узел.

Vector — реализация динамического массива объектов, методы которой синхронизированы.

Stack — реализация стека LIFO (last-in-first-out).

Set (сет) описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Реализации:

HashSet - использует HashMap для хранения данных. В качестве ключа и значения используется добавляемый элемент. Из-за особенностей реализации порядок элементов не гарантируется при добавлении.

LinkedHashSet — гарантирует, что порядок элементов при обходе коллекции будет идентичен порядку добавления элементов.

TreeSet — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».

Queue (очередь) предназначена для хранения элементов с предопределённым способом вставки и извлечения FIFO (first-in-first-out):

PriorityQueue — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».

ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out).

Интерфейс Map реализован классами:

Hashtable — хэш-таблица, методы которой синхронизированы. Не позволяет использовать null в качестве значения или ключа и не является упорядоченной.

HashMap — хэш-таблица. Позволяет использовать null в качестве значения или ключа и не является упорядоченной.

LinkedHashMap — упорядоченная реализация хэш-таблицы.

TreeMap — реализация, основанная на красно-чёрных деревьях. Является упорядоченной и предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».

WeakHashMap — реализация хэш-таблицы, которая организована с использованием weak references для ключей (сборщик мусора автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элемента нет жёстких ссылок).

Расположите в виде иерархии следующие интерфейсы: List, Set, Map, SortedSet, SortedMap, Collection, Iterable, Iterator, NavigableSet, NavigableMap.

Iterable

Collection

List

Set

SortedSet

NavigableSet

Map

SortedMap

NavigableMap

Iterator

**Почему Map — это не Collection, в то время как List и Set являются Collection?**

Collection представляет собой совокупность некоторых элементов. Map - это совокупность пар «ключ-значение».

**В чем разница между классами java.util.Collection и java.util.Collections?**

java.util.Collections - набор статических методов для работы с коллекциями.

java.util.Collection - один из основных интерфейсов Java Collections Framework.

**Что такое «fail-fast поведение»?**

fail-fast поведение означает, что при возникновении ошибки или состояния, которое может привести к ошибке, система немедленно прекращает дальнейшую работу и уведомляет об этом. Использование fail-fast подхода позволяет избежать недетерминированного поведения программы в течение времени.

В Java Collections API некоторые итераторы ведут себя как fail-fast и выбрасывают ConcurrentModificationException, если после его создания была произведена модификация коллекции, т.е. добавлен или удален элемент напрямую из коллекции, а не используя методы итератора.

Реализация такого поведения осуществляется за счет подсчета количества модификаций коллекции (modification count):

при изменении коллекции счетчик модификаций так же изменяется;

при создании итератора ему передается текущее значение счетчика;

при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение.

**Какая разница между fail-fast и fail-safe?**

В противоположность fail-fast, итераторы fail-safe не вызывают никаких исключений при изменении структуры, потому что они работают с клоном коллекции вместо оригинала.

Приведите примеры итераторов, реализующих поведение fail-safe

Итератор коллекции CopyOnWriteArrayList и итератор представления keySet коллекции ConcurrentHashMap являются примерами итераторов fail-safe.

**Чем различаются Enumeration и Iterator.**

Хотя оба интерфейса и предназначены для обхода коллекций между ними имеются существенные различия:

с помощью Enumeration нельзя добавлять/удалять элементы;

в Iterator исправлены имена методов для повышения читаемости кода (Enumeration.hasMoreElements() соответствует Iterator.hasNext(), Enumeration.nextElement() соответствует Iterator.next() и т.д);

Enumeration присутствуют в устаревших классах, таких как Vector/Stack, тогда как Iterator есть во всех современных классах-коллекциях.

**Как между собой связаны Iterable и Iterator?**

Интерфейс Iterable имеет только один метод - iterator(), который возвращает Iterator.

**Как между собой связаны Iterable, Iterator и «for-each»?**

Классы, реализующие интерфейс Iterable, могут применяться в конструкции for-each, которая использует Iterator.

**Сравните Iterator и ListIterator.**

ListIterator расширяет интерфейс Iterator

ListIterator может быть использован только для перебора элементов коллекции List;

Iterator позволяет перебирать элементы только в одном направлении, при помощи метода next(). Тогда как ListIterator позволяет перебирать список в обоих направлениях, при помощи методов next() и previous();

ListIterator не указывает на конкретный элемент: его текущая позиция располагается между элементами, которые возвращают методы previous() и next().

При помощи ListIterator вы можете модифицировать список, добавляя/удаляя элементы с помощью методов add() и remove(). Iterator не поддерживает данного функционала.

**Что произойдет при вызове Iterator.next() без предварительного вызова Iterator.hasNext()?**

Если итератор указывает на последний элемент коллекции, то возникнет исключение NoSuchElementException, иначе будет возвращен следующий элемент.

**Сколько элементов будет пропущено, если Iterator.next() будет вызван после 10-ти вызовов Iterator.hasNext()?**

Нисколько - hasNext() осуществляет только проверку наличия следующего элемента.

**Как поведёт себя коллекция, если вызвать iterator.remove()?**

Если вызову iterator.remove() предшествовал вызов iterator.next(), то iterator.remove() удалит элемент коллекции, на который указывает итератор, в противном случае будет выброшено IllegalStateException().

**Как поведёт себя уже инстанциированный итератор для collection, если вызвать collection.remove()?**

При следующем вызове методов итератора будет выброшено ConcurrentModificationException.

**Как избежать ConcurrentModificationException во время перебора коллекции?**

Попробовать подобрать или реализовать самостоятельно другой итератор, работающий по принципу fail-safe.

Использовать ConcurrentHashMap и CopyOnWriteArrayList.

Преобразовать список в массив и перебирать массив.

Блокировать изменения списка на время перебора с помощью блока synchronized.

Отрицательная сторона последних двух вариантов - ухудшение производительности.

**Какая коллекция реализует дисциплину обслуживания FIFO?**

FIFO, First-In-First-Out («первым пришел-первым ушел») - по этому принципу построена коллекция Queue.

**Какая коллекция реализует дисциплину обслуживания FILO?**

FILO, First-In-Last-Out («первым пришел, последним ушел») - по этому принципу построена коллекция Stack.

**Зачем добавили ArrayList, если уже был Vector?**

Методы класса Vector синхронизированы, а ArrayList - нет;

По умолчанию, Vector удваивает свой размер, когда заканчивается выделенная под элементы память. ArrayList же увеличивает свой размер только на половину.

Vector это устаревший класс и его использование не рекомендовано.

**Чем отличается ArrayList от LinkedList? В каких случаях лучше использовать первый, а в каких второй?**

ArrayList это список, реализованный на основе массива, а LinkedList — это классический двусвязный список, основанный на объектах с ссылками между ними.

ArrayList:

доступ к произвольному элементу по индексу за константное время O(1);

доступ к элементам по значению за линейное время O(N);

вставка в конец в среднем производится за константное время O(1);

удаление произвольного элемента из списка занимает значительное время т.к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку влево (реальный размер массива (capacity) не изменяется);

вставка элемента в произвольное место списка занимает значительное время т.к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку вправо;

минимум накладных расходов при хранении.

LinkedList:

на получение элемента по индексу или значению потребуется линейное время O(N);

но доступ к первому и последнему элементу списка всегда осуществляется за константное время O(1) — ссылки постоянно хранятся на первый и последний элемент;

на добавление и удаление в начало или конец списка потребуется константное O(1);

вставка или удаление в/из произвольного место константное O(1);

но поиск позиции вставки и удаления за линейное время O(N);

требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка.

В целом, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти, и по скорости выполнения операций. LinkedList предпочтительно применять, когда нужны частые операции вставки/удаления или в случаях, когда необходимо гарантированное время добавления элемента в список.

**Что работает быстрее ArrayList или LinkedList?**

Смотря какие действия будут выполняться над структурой.

**Какое худшее время работы метода contains() для элемента, который есть в LinkedList?**

O(N). Время поиска элемента линейно пропорционально количеству элементов в списке.

**Какое худшее время работы метода contains() для элемента, который есть в ArrayList?**

O(N). Время поиска элемента линейно пропорционально количеству элементов с списке.

**Какое худшее время работы метода add() для LinkedList?**

O(N). Добавление в начало/конец списка осуществляется за время O(1).

**Какое худшее время работы метода add() для ArrayList?**

O(N). Вставка элемента в конец списка осуществляется за время O(1), но если вместимость массива недостаточна, то происходит создание нового массива с увеличенным размером и копирование всех элементов из старого массива в новый.

**Необходимо добавить 1 млн. элементов, какую структуру вы используете?**

Однозначный ответ можно дать только исходя из информации о том в какую часть списка происходит добавление элементов, что потом будет происходить с элементами списка, существуют ли какие-то ограничения по памяти или скорости выполнения.

**Как происходит удаление элементов из ArrayList? Как меняется в этом случае размер ArrayList?**

При удалении произвольного элемента из списка, все элементы, находящиеся «правее» смещаются на одну ячейку влево и реальный размер массива (его емкость, capacity) не изменяется никак. Механизм автоматического «расширения» массива существует, а вот автоматического «сжатия» нет, можно только явно выполнить «сжатие» командой trimToSize().

Предложите эффективный алгоритм удаления нескольких рядом стоящих элементов из середины списка, реализуемого ArrayList.

Допустим нужно удалить n элементов с позиции m в списке. Вместо выполнения удаления одного элемента n раз (каждый раз смещая на 1 позицию элементы, стоящие «правее» в списке), нужно выполнить смещение всех элементов, стоящих «правее» n + m позиции на n элементов «левее» к началу списка. Таким образом, вместо выполнения n итераций перемещения элементов списка, все выполняется за 1 проход. Но если говорить об общей эффективности - то самый быстрый способ будет с использованием System.arraycopy(), и получить к нему доступ можно через метод - subList(int fromIndex, int toIndex)

Пример:

import java.io.\*;

import java.util.ArrayList;

public class Main {

//позиция, с которой удаляем

private static int m = 0;

//количество удаляемых элементов

private static int n = 0;

//количество элементов в списке

private static final int size = 1000000;

//основной список (для удаления вызовом remove() и его копия для удаления путём перезаписи)

private static ArrayList<Integer> initList, copyList;

public static void main(String[] args){

initList = new ArrayList<>(size);

for (int i = 0; i < size; i++) {

initList.add(i);

}

System.out.println("Список из 1.000.000 элементов заполнен");

copyList = new ArrayList<>(initList);

System.out.println("Создана копия списка\n");

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

try{

System.out.print("С какой позиции удаляем? > ");

m = Integer.parseInt(br.readLine());

System.out.print("Сколько удаляем? > ");

n = Integer.parseInt(br.readLine());

} catch(IOException e){

System.err.println(e.toString());

}

System.out.println("\nВыполняем удаление вызовом remove()...");

long start = System.currentTimeMillis();

for (int i = m - 1; i < m + n - 1; i++) {

initList.remove(i);

}

long finish = System.currentTimeMillis() - start;

System.out.println("Время удаления с помощью вызова remove(): " + finish);

System.out.println("Размер исходного списка после удаления: " + initList.size());

System.out.println("\nВыполняем удаление путем перезаписи...\n");

start = System.currentTimeMillis();

removeEfficiently();

finish = System.currentTimeMillis() - start;

System.out.println("Время удаления путём смещения: " + finish);

System.out.println("Размер копии списка:" + copyList.size());

System.out.println("\nВыполняем удаление через SubList...\n");

start = System.currentTimeMillis();

initList.subList(m - 1, m + n).clear();

finish = System.currentTimeMillis() - start;

System.out.println("Время удаления через саблист: " + finish);

System.out.println("Размер копии списка:" + copyList.size());

}

private static void removeEfficiently(){

/\* если необходимо удалить все элементы, начиная с указанного,

\* то удаляем элементы с конца до m

\*/

if (m + n >= size){

int i = size - 1;

while (i != m - 1){

copyList.remove(i);

i--;

}

} else{

//переменная k необходима для отсчёта сдвига начиная от места вставка m

for (int i = m + n, k = 0; i < size; i++, k++) {

copyList.set(m + k, copyList.get(i));

}

/\* удаляем ненужные элементы в конце списка

\* удаляется всегда последний элемент, так как время этого действия

\* фиксировано и не зависит от размера списка

\*/

int i = size - 1;

while (i != size - n - 1){

copyList.remove(i);

i--;

}

//сокращаем длину списка путём удаления пустых ячеек

copyList.trimToSize();

}

}

}

Результат выполнения:

run:

Список из 1.000.000 элементов заполнен

Создана копия списка

С какой позиции удаляем? > 600000

Сколько удаляем? > 20000

Выполняем удаление вызовом remove()...

Время удаления с помощью вызова remove(): 928

Размер исходного списка после удаления: 980000

Выполняем удаление путем перезаписи...

Время удаления путём смещения: 17

Размер копии списка:980000

Выполняем удаление через SubList...

Время удаления через саблист: 1

Размер копии списка:980000

СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 33 секунды)

Сколько необходимо дополнительной памяти при вызове ArrayList.add()?

Если в массиве достаточно места для размещения нового элемента, то дополнительной памяти не требуется. Иначе происходит создание нового массива размером в 1,5 раза превышающим существующий (это верно для JDK выше 1.7, в более ранних версиях размер увеличения иной).

Сколько выделяется дополнительно памяти при вызове LinkedList.add()?

Создается один новый экземпляр вложенного класса Node.

Оцените количество памяти на хранение одного примитива типа byte в LinkedList?

Каждый элемент LinkedList хранит ссылку на предыдущий элемент, следующий элемент и ссылку на данные.

private static class Node<E> {

E item;

Node<E> next;

Node<E> prev;

//...

}

Для 32-битных систем каждая ссылка занимает 32 бита (4 байта). Сам объект (заголовок) вложенного класса Node занимает 8 байт. 4 + 4 + 4 + 8 = 20 байт, а т.к. размер каждого объекта в Java кратен 8, соответственно получаем 24 байта. Примитив типа byte занимает 1 байт памяти, но в JCF примитивы упаковываются: объект типа Byte занимает в памяти 16 байт (8 байт на заголовок объекта, 1 байт на поле типа byte и 7 байт для кратности 8). Также напомню, что значения от -128 до 127 кэшируются и для них новые объекты каждый раз не создаются. Таким образом, в x32 JVM 24 байта тратятся на хранение одного элемента в списке и 16 байт - на хранение упакованного объекта типа Byte. Итого 40 байт.

Для 64-битной JVM каждая ссылка занимает 64 бита (8 байт), размер заголовка каждого объекта составляет 16 байт (два машинных слова). Вычисления аналогичны: 8 + 8 + 8 + 16 = 40байт и 24 байта. Итого 64 байта.

**Оцените количество памяти на хранение одного примитива типа byte в ArrayList?**

ArrayList основан на массиве, для примитивных типов данных осуществляется автоматическая упаковка значения, поэтому 16 байт тратится на хранение упакованного объекта и 4 байта (8 для x64) - на хранение ссылки на этот объект в самой структуре данных. Таким образом, в x32 JVM 4 байта используются на хранение одного элемента и 16 байт - на хранение упакованного объекта типа Byte. Для x64 - 8 байт и 24 байта соответственно.

Для ArrayList или для LinkedList операция добавления элемента в середину (list.add(list.size()/2, newElement)) медленнее?

Для ArrayList:

проверка массива на вместимость. Если вместимости недостаточно, то увеличение размера массива и копирование всех элементов в новый массив (O(N));

копирование всех элементов, расположенных правее от позиции вставки, на одну позицию вправо (O(N));

вставка элемента (O(1)).

Для LinkedList:

поиск позиции вставки (O(N));

вставка элемента (O(1)).

В худшем случае вставка в середину списка эффективнее для LinkedList. В остальных - скорее всего, для ArrayList, поскольку копирование элементов осуществляется за счет вызова быстрого системного метода System.arraycopy().

В реализации класса ArrayList есть следующие поля: Object[] elementData, int size. Объясните, зачем хранить отдельно size, если всегда можно взять elementData.length?

Размер массива elementData представляет собой вместимость (capacity) ArrayList, которая всегда больше переменной size - реального количества хранимых элементов. При необходимости вместимость автоматически возрастает.

**Кто кого расширяет: Queue расширяет Deque, или Deque расширяет Queue?**

Queue - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди. Хотя этот принцип нарушает, к примеру, PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

Deque (Double Ended Queue) расширяет Queue и согласно документации, это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого, реализации интерфейса Deque могут строится по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

**Почему LinkedList реализует и List, и Deque?**

LinkedList позволяет добавлять элементы в начало и конец списка за константное время, что хорошо согласуется с поведением интерфейса Deque.

**LinkedList — это односвязный, двусвязный или четырехсвязный список?**

Двусвязный: каждый элемент LinkedList хранит ссылку на предыдущий и следующий элементы.

**Как перебрать элементы LinkedList в обратном порядке, не используя медленный get(index)?**

Для этого в LinkedList есть обратный итератор, который можно получить вызва метод descendingIterator().

**Что позволяет сделать PriorityQueue?**

Особенностью PriorityQueue является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

Используя PriorityQueue, можно, например, реализовать алгоритм Дейкстры для поиска кратчайшего пути от одной вершины графа к другой. Либо для хранения объектов согласно определённого свойства.

**Stack считается «устаревшим». Чем его рекомендуют заменять? Почему?**

Stack был добавлен в Java 1.0 как реализация стека LIFO (last-in-first-out) и является расширением коллекции Vector, хотя это несколько нарушает понятие стека (например, класс Vector предоставляет возможность обращаться к любому элементу по индексу). Является частично синхронизированной коллекцией (кроме метода добавления push()) с вытекающими отсюда последствиями в виде негативного воздействия на производительность. После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать реализации именно этого интерфейса, например, ArrayDeque.

**Зачем нужен HashMap, если есть Hashtable?**

Методы класса Hashtable синхронизированы, что приводит к снижению производительности, а HashMap - нет;

HashTable не может содержать элементы null, тогда как HashMap может содержать один ключ null и любое количество значений null;

Iterator у HashMap, в отличие от Enumeration у HashTable, работает по принципу «fail-fast» (выдает исключение при любой несогласованности данных).

Hashtable это устаревший класс и его использование не рекомендовано.

**В чем разница между HashMap и IdentityHashMap? Для чего нужна IdentityHashMap?**

IdentityHashMap - это структура данных, так же реализующая интерфейс Map и использующая при сравнении ключей (значений) сравнение ссылок, а не вызов метода equals(). Другими словами, в IdentityHashMap два ключа k1 и k2 будут считаться равными, если они указывают на один объект, т.е. выполняется условие k1 == k2.

IdentityHashMap не использует метод hashCode(), вместо которого применяется метод System.identityHashCode(), по этой причине IdentityHashMap по сравнению с HashMap имеет более высокую производительность, особенно если последний хранит объекты с дорогостоящими методами equals() и hashCode().

Одним из основных требований к использованию HashMap является неизменяемость ключа, а, т.к. IdentityHashMap не использует методы equals() и hashCode(), то это правило на него не распространяется.

IdentityHashMap может применяться для реализации сериализации/клонирования. При выполнении подобных алгоритмов программе необходимо обслуживать хэш-таблицу со всеми ссылками на объекты, которые уже были обработаны. Такая структура не должна рассматривать уникальные объекты как равные, даже если метод equals() возвращает true.

Пример кода:

import java.util.HashMap;

import java.util.IdentityHashMap;

import java.util.Map;

public class Q2 {

public static void main(String[] args) {

Q2 q = new Q2();

q.testHashMapAndIdentityHashMap();

}

private void testHashMapAndIdentityHashMap() {

CreditCard visa = new CreditCard("VISA", "04/12/2019");

Map<CreditCard, String> cardToExpiry = new HashMap<>();

Map<CreditCard, String> cardToExpiryIdenity = new IdentityHashMap<>();

System.out.println("adding to HM");

// inserting objects to HashMap

cardToExpiry.put(visa, visa.getExpiryDate());

// inserting objects to IdentityHashMap

cardToExpiryIdenity.put(visa, visa.getExpiryDate());

System.out.println("adding to IHM");

System.out.println("before modifying keys");

String result = cardToExpiry.get(visa) != null ? "Yes" : "No";

System.out.println("Does VISA card exists in HashMap? " + result);

result = cardToExpiryIdenity.get(visa) != null ? "Yes" : "No";

System.out.println("Does VISA card exists in IdenityHashMap? " + result);

// modifying value object

visa.setExpiryDate("02/11/2030");

System.out.println("after modifying keys");

result = cardToExpiry.get(visa) != null ? "Yes" : "No";

System.out.println("Does VISA card exists in HashMap? " + result);

result = cardToExpiryIdenity.get(visa) != null ? "Yes" : "No";

System.out.println("Does VISA card exists in IdenityHashMap? " + result);

System.out.println("cardToExpiry.containsKey");

System.out.println(cardToExpiry.containsKey(visa));

System.out.println("cardToExpiryIdenity.containsKey");

System.out.println(cardToExpiryIdenity.containsKey(visa));

}

}

class CreditCard {

private String issuer;

private String expiryDate;

public CreditCard(String issuer, String expiryDate) {

this.issuer = issuer;

this.expiryDate = expiryDate;

}

public String getIssuer() {

return issuer;

}

public String getExpiryDate() {

return expiryDate;

}

public void setExpiryDate(String expiry) {

this.expiryDate = expiry;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((expiryDate == null) ? 0 : expiryDate.hashCode());

result = prime \* result + ((issuer == null) ? 0 : issuer.hashCode());

System.out.println("hashCode = " + result);

return result;

}

@Override

public boolean equals(Object obj) {

System.out.println("equals !!! ");

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

CreditCard other = (CreditCard) obj;

if (expiryDate == null) {

if (other.expiryDate != null)

return false;

} else if (!expiryDate.equals(other.expiryDate))

return false;

if (issuer == null) {

if (other.issuer != null)

return false;

} else if (!issuer.equals(other.issuer))

return false;

return true;

}

}

Результат выполнения кода:

adding to HM

hashCode = 1285631513

adding to IHM

before modifying keys

hashCode = 1285631513

Does VISA card exists in HashMap? Yes

Does VISA card exists in IdenityHashMap? Yes

after modifying keys

hashCode = 791156485

Does VISA card exists in HashMap? No

Does VISA card exists in IdenityHashMap? Yes

cardToExpiry.containsKey

hashCode = 791156485

false

cardToExpiryIdenity.containsKey

true

**В чем разница между HashMap и WeakHashMap? Для чего используется WeakHashMap?**

В Java существует 4 типа ссылок: сильные (strong reference), мягкие (SoftReference), слабые (WeakReference) и фантомные (PhantomReference). Особенности каждого типа ссылок связаны с работой Garbage Collector. Если объект можно достичь только с помощью цепочки WeakReference (то есть на него отсутствуют сильные и мягкие ссылки), то данный объект будет помечен на удаление.

WeakHashMap - это структура данных, реализующая интерфейс Map и основанная на использовании WeakReference для хранения ключей. Таким образом, пара «ключ-значение» будет удалена из WeakHashMap, если на объект-ключ более не имеется сильных ссылок.

В качестве примера использования такой структуры данных можно привести следующую ситуацию: допустим имеются объекты, которые необходимо расширить дополнительной информацией, при этом изменение класса этих объектов нежелательно либо невозможно. В этом случае добавляем каждый объект в WeakHashMap в качестве ключа, а в качестве значения - нужную информацию. Таким образом, пока на объект имеется сильная ссылка (либо мягкая), можно проверять хэш-таблицу и извлекать информацию. Как только объект будет удален, то WeakReference для этого ключа будет помещен в ReferenceQueue и затем соответствующая запись для этой слабой ссылки будет удалена из WeakHashMap.

В WeakHashMap используются WeakReferences. А почему бы не создать SoftHashMap на SoftReferences?

SoftHashMap представлена в сторонних библиотеках, например, в Apache Commons.

**В WeakHashMap используются WeakReferences. А почему бы не создать PhantomHashMap на PhantomReferences?**

PhantomReference при вызове метода get() возвращает всегда null, поэтому тяжело представить назначение такой структуры данных.

**LinkedHashMap - что в нем от LinkedList, а что от HashMap?**

Реализация LinkedHashMap отличается от HashMap поддержкой двухсвязанного списка, определяющего порядок итерации по элементам структуры данных. По умолчанию элементы списка упорядочены согласно их порядку добавления в LinkedHashMap (insertion-order). Однако порядок итерации можно изменить, установив параметр конструктора accessOrder в значение true. В этом случае доступ осуществляется по порядку последнего обращения к элементу (access-order). Это означает, что при вызове методов get() или put() элемент, к которому обращаемся, перемещается в конец списка.

При добавлении элемента, который уже присутствует в LinkedHashMap (т.е. с одинаковым ключом), порядок итерации по элементам не изменяется.

**В чем проявляется «сортированность» SortedMap, кроме того, что toString() выводит все элементы по порядку?**

Так же оно проявляется при итерации по коллекции.

**Как устроен HashMap?**

HashMap состоит из «корзин» (bucket). С технической точки зрения «корзины» — это элементы массива, которые хранят ссылки на списки элементов. При добавлении новой пары «ключ-значение», вычисляет хэш-код ключа, на основании которого вычисляется номер корзины (номер ячейки массива), в которую попадет новый элемент. Если корзина пустая, то в нее сохраняется ссылка на вновь добавляемый элемент, если же там уже есть элемент, то происходит последовательный переход по ссылкам между элементами в цепочке, в поисках последнего элемента, от которого и ставится ссылка на вновь добавленный элемент. Если в списке был найден элемент с таким же ключом, то он заменяется.

Согласно Кнуту и Кормену существует две основных реализации хэш-таблицы: на основе открытой адресации и на основе метода цепочек. Как реализована HashMap? Почему, по вашему мнению, была выбрана именно эта реализация? В чем плюсы и минусы каждого подхода?

HashMap реализован с использованием метода цепочек, т.е. каждой ячейке массива (корзине) соответствует свой связный список и при возникновении коллизии осуществляется добавление нового элемента в этот список.

Для метода цепочек коэффициент заполнения может быть больше 1 и с увеличением числа элементов производительность убывает линейно. Такие таблицы удобно использовать, если заранее неизвестно количество хранимых элементов, либо их может быть достаточно много, что приводит к большим значениям коэффициента заполнения.

Среди методов открытой реализации различают:

линейное пробирование;

квадратичное пробирование;

двойное хэширование.

Недостатки структур с методом открытой адресации:

Количество элементов в хэш-таблице не может превышать размера массива. По мере увеличения числа элементов и повышения коэффициента заполнения производительность структуры резко падает, поэтому необходимо проводить перехэширование.

Сложно организовать удаление элемента.

Первые два метода открытой адресации приводят к проблеме первичной и вторичной группировок.

Преимущества хэш-таблицы с открытой адресацией:

отсутствие затрат на создание и хранение объектов списка;

простота организации сериализации/десериализации объекта.

**Как работает HashMap при попытке сохранить в него два элемента по ключам с одинаковым hashCode(), но для которых equals() == false?**

По значению hashCode() вычисляется индекс ячейки массива, в список которой этот элемент будет добавлен. Перед добавлением осуществляется проверка на наличие элементов в этой ячейке. Если элементы с таким hashCode() уже присутствует, но их equals() методы не равны, то элемент будет добавлен в конец списка.

**Какое начальное количество корзин в HashMap?**

В конструкторе по умолчанию - 16, используя конструкторы с параметрами можно задавать произвольное начальное количество корзин.

Какова оценка временной сложности операций над элементами из HashMap? Гарантирует ли HashMap указанную сложность выборки элемента?

В общем случае операции добавления, поиска и удаления элементов занимают константное время.

Данная сложность не гарантируется, т.к. если хэш-функция распределяет элементы по корзинам равномерно, временная сложность станет не хуже Логарифмического времени O(log(N)), а в случае, когда хэш-функция постоянно возвращает одно и то же значение, HashMap превратится в связный список со сложностью О(n).

Пример кода двоичного поиска:

public class Q {

public static void main(String[] args) {

Q q = new Q();

q.binSearch();

}

private void binSearch() {

int[] inpArr = {1, 2, 3, 4, 5, 6, 7, 8, 9};

Integer result = binSearchF(inpArr, 1, 0, inpArr.length - 1);

System.out.println("-----------------------");

result = binSearchF(inpArr, 2, 0, inpArr.length - 1);

System.out.println("Found at position " + result);

}

private Integer binSearchF(int[] inpArr, int searchValue, int low, int high) {

Integer index = null;

while (low <= high) {

System.out.println("New iteration, low = " + low + ", high = " + high);

int mid = (low + high) / 2;

System.out.println("trying mid = " + mid + " inpArr[mid] = " + inpArr[mid]);

if (inpArr[mid] < searchValue) {

low = mid + 1;

System.out.println("inpArr[mid] (" + inpArr[mid] + ") < searchValue(" + searchValue + "), mid = " + mid

+ ", setting low = " + low);

} else if (inpArr[mid] > searchValue) {

high = mid - 1;

System.out.println("inpArr[mid] (" + inpArr[mid] + ") > searchValue(" + searchValue + "), mid = " + mid

+ ", setting high = " + high);

} else if (inpArr[mid] == searchValue) {

index = mid;

System.out.println("found at index " + mid);

break;

}

}

return index;

}

}

**Возможна ли ситуация, когда HashMap выродится в список даже с ключами имеющими разные hashCode()?**

Это возможно в случае, если метод, определяющий номер корзины будет возвращать одинаковые значения.

**В каком случае может быть потерян элемент в HashMap?**

Допустим, в качестве ключа используется не примитив, а объект с несколькими полями. После добавления элемента в HashMap у объекта, который выступает в качестве ключа, изменяют одно поле, которое участвует в вычислении хэш-кода. В результате при попытке найти данный элемент по исходному ключу, будет происходить обращение к правильной корзине, а вот equals уже не найдет указанный ключ в списке элементов. Тем не менее, даже если equals реализован таким образом, что изменение данного поля объекта не влияет на результат, то после увеличения размера корзин и пересчета хэш-кодов элементов, указанный элемент, с измененным значением поля, с большой долей вероятности попадет в совершенно другую корзину и тогда уже потеряется совсем.

**Почему нельзя использовать byte[] в качестве ключа в HashMap?**

Хэш-код массива не зависит от хранимых в нем элементов, а присваивается при создании массива (метод вычисления хэш-кода массива не переопределен и вычисляется по стандартному Object.hashCode() на основании адреса массива). Так же у массивов не переопределен equals и выполняется сравнение указателей. Это приводит к тому, что обратиться к сохраненному с ключом-массивом элементу не получится при использовании другого массива такого же размера и с такими же элементами, доступ можно осуществить лишь в одном случае — при использовании той же самой ссылки на массив, что использовалась для сохранения элемента.

**Какова роль equals() и hashCode() в HashMap?**

hashCode позволяет определить корзину для поиска элемента, а equals используется для сравнения ключей элементов в списке корзины и искомого ключа.

**Каково максимальное число значений hashCode()?**

Число значений следует из сигнатуры int hashCode() и равно диапазону типа int — 232.

**Какое худшее время работы метода get(key) для ключа, который есть в HashMap?**

O(N). Худший случай - это поиск ключа в HashMap, вырожденного в список по причине совпадения ключей по hashCode() и для выяснения хранится ли элемент с определённым ключом может потребоваться перебор всего списка.

Но начиная с Java 8, после определенного числа элементов в списке, связный список преобразовывается в красно-черное дерево и сложность выборки, даже в случае плохой хеш-функции, не хуже логарифмической O(log(N))

**Сколько переходов происходит в момент вызова HashMap.get(key) по ключу, который есть в таблице?**

ключ равен null: 1 - выполняется единственный метод getForNullKey().

любой ключ отличный от null: 4 - вычисление хэш-кода ключа; определение номера корзины; поиск значения; возврат значения.

**Сколько создается новых объектов, когда вы добавляете новый элемент в HashMap?**

Один новый объект статического вложенного класса Entry<K,V>.

**Как и когда происходит увеличение количества корзин в HashMap?**

Помимо capacity у HashMap есть еще поле loadFactor, на основании которого, вычисляется предельное количество занятых корзин capacity \* loadFactor. По умолчанию loadFactor = 0.75. По достижению предельного значения, число корзин увеличивается в 2 раза и для всех хранимых элементов вычисляется новое «местоположение» с учетом нового числа корзин.

Объясните смысл параметров в конструкторе HashMap(int initialCapacity, float loadFactor).

initialCapacity - исходный размер HashMap, количество корзин в хэш-таблице в момент её создания.

loadFactor - коэффициент заполнения HashMap, при превышении которого происходит увеличение количества корзин и автоматическое перехэширование. Равен отношению числа уже хранимых элементов в таблице к её размеру.

**Будет ли работать HashMap, если все добавляемые ключи будут иметь одинаковый hashCode()?**

Да, будет, но в этом случае HashMap вырождается в связный список и теряет свои преимущества.

**Как перебрать все ключи Map?**

Использовать метод keySet(), который возвращает множество Set<K> ключей.

**Как перебрать все значения Map?**

Использовать метод values(), который возвращает коллекцию Collection<V> значений.

**Как перебрать все пары «ключ-значение» в Map?**

Использовать метод entrySet(), который возвращает множество Set<Map.Entry<K, V> пар «ключ-значение».

**В чем отличия TreeSet и HashSet?**

TreeSet обеспечивает упорядоченно хранение элементов в виде красно-черного дерева. Сложность выполнения основных операций не хуже O(log(N)) (Логарифмическое время).

HashSet использует для хранения элементов такой же подход, что и HashMap, за тем отличием, что в HashSet в качестве ключа и значения выступает сам элемент, кроме того, HashSet не поддерживает упорядоченное хранение элементов и обеспечивает временную сложность выполнения операций аналогично HashMap.

**Что будет, если добавлять элементы в TreeSet по возрастанию?**

В основе TreeSet лежит красно-черное дерево, которое умеет само себя балансировать. В итоге, TreeSet все равно в каком порядке вы добавляете в него элементы, преимущества этой структуры данных будут сохраняться.

**Чем LinkedHashSet отличается от HashSet?**

LinkedHashSet отличается от HashSet только тем, что в его основе лежит LinkedHashMap вместо HashMap. Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов (insertion-order). При добавлении элемента, который уже присутствует в LinkedHashSet (т.е. с одинаковым ключом), порядок обхода элементов не изменяется.

**Для Enum есть специальный класс java.util.EnumSet. Зачем? Чем авторов не устраивал HashSet или TreeSet?**

EnumSet - это реализация интерфейса Set для использования с перечислениями (Enum). В структуре данных хранятся объекты только одного типа Enum, указываемого при создании. Для хранения значений EnumSet использует массив битов (bit vector), - это позволяет получить высокую компактность и эффективность. Проход по EnumSet осуществляется согласно порядку объявления элементов перечисления.

Все основные операции выполняются за O(1) и обычно (но негарантированно) быстрей аналогов из HashSet, а пакетные операции (bulk operations), такие как containsAll() и retainAll() выполняются даже горазда быстрей.

Помимо всего EnumSet предоставляет множество статических методов инициализации для упрощенного и удобного создания экземпляров.

**Какие существуют способы перебирать элементы списка?**

Цикл с итератором

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

//iterator.next();

}

Цикл for

for (int i = 0; i < list.size(); i++) {

//list.get(i);

}

Цикл while

int i = 0;

while (i < list.size()) {

//list.get(i);

i++;

}

«for-each»

for (String element : list) {

//element;

}

**Каким образом можно получить синхронизированные объекты стандартных коллекций?**

С помощью статических методов synchronizedMap() и synchronizedList() класса Collections. Данные методы возвращают синхронизированный декоратор переданной коллекции. При этом все равно в случае обхода по коллекции требуется ручная синхронизация.

Map m = Collections.synchronizedMap(new HashMap());

List l = Collections.synchronizedList(new ArrayList());

Начиная с Java 6 JCF был расширен специальными коллекциями, поддерживающими многопоточный доступ, такими как CopyOnWriteArrayList и ConcurrentHashMap.

**Как получить коллекцию только для чтения?**

При помощи:

Collections.unmodifiableList(list);

Collections.unmodifiableSet(set);

Collections.unmodifiableMap(map).

Эти методы принимают коллекцию в качестве параметра, и возвращают коллекцию только для чтения с теми же элементами внутри.

**Напишите однопоточную программу, которая заставляет коллекцию выбросить ConcurrentModificationException.**

public static void main(String[] args) {

List<Integer> list = new ArrayList<>();

list.add(1);

list.add(2);

list.add(3);

for (Integer integer : list) {

list.remove(1);

}

}

**Приведите пример, когда какая-либо коллекция выбрасывает UnsupportedOperationException.**

public static void main(String[] args) {

List<Integer> list = Collections.emptyList();

list.add(0);

}

**Реализуйте симметрическую разность двух коллекций используя методы Collection (addAll(...), removeAll(...), retainAll(...)).**

Симметрическая разность двух коллекций - это множество элементов, одновременно не принадлежащих обоим исходным коллекциям.

<T> Collection<T> symmetricDifference(Collection<T> a, Collection<T> b) {

// Объединяем коллекции.

Collection<T> result = new ArrayList<>(a);

result.addAll(b);

// Получаем пересечение коллекций.

Collection<T> intersection = new ArrayList<>(a);

intersection.retainAll(b);

// Удаляем элементы, расположенные в обоих коллекциях.

result.removeAll(intersection);

return result;

}

**Как, используя LinkedHashMap, сделать кэш c «invalidation policy»?**

Необходимо использовать LRU-алгоритм (Least Recently Used algorithm) и LinkedHashMap с access-order. В этом случае при обращении к элементу он будет перемещаться в конец списка, а наименее используемые элементы будут постепенно группироваться в начале списка. Так же в стандартной реализации LinkedHashMap есть метод removeEldestEntries(), который возвращает true, если текущий объект LinkedHashMap должен удалить наименее используемый элемент из коллекции при использовании методов put() и putAll().

public class LRUCache<K, V> extends LinkedHashMap<K, V> {

private static final int MAX\_ENTRIES = 10;

public LRUCache(int initialCapacity) {

super(initialCapacity, 0.85f, true);

}

@Override

protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {

return size() > MAX\_ENTRIES;

}

}

Стоит заметить, что LinkedHashMap не позволяет полностью реализовать LRU-алгоритм, поскольку при вставке уже имеющегося в коллекции элемента порядок итерации по элементам не меняется.

**Как одной строчкой скопировать элементы любой collection в массив?**

Object[] array = collection.toArray();

**Как одним вызовом из List получить List со всеми элементами, кроме первых и последних 3-х?**

List<Integer> subList = list.subList(3, list.size() - 3);

Как одной строчкой преобразовать HashSet в ArrayList?

ArrayList<Integer> list = new ArrayList<>(new HashSet<>());

Как одной строчкой преобразовать ArrayList в HashSet?

HashSet<Integer> set = new HashSet<>(new ArrayList<>());

Сделайте HashSet из ключей HashMap.

HashSet<Object> set = new HashSet<>(map.keySet());

Сделайте HashMap из HashSet<Map.Entry<K, V>>.

HashMap<K, V> map = new HashMap<>(set.size());

for (Map.Entry<K, V> entry : set) {

map.put(entry.getKey(), entry.getValue());

}

**Какие нововведения, появились в Java 8 и JDK 8?**

Методы интерфейсов по умолчанию;

Лямбда-выражения;

Функциональные интерфейсы;

Ссылки на методы и конструкторы;

Повторяемые аннотации;

Аннотации на типы данных;

Рефлексия для параметров методов;

Stream API для работы с коллекциями;

Параллельная сортировка массивов;

Новое API для работы с датами и временем;

Новый движок JavaScript Nashorn;

Добавлено несколько новых классов для потокобезопасной работы;

Добавлен новый API для Calendar и Locale;

Добавлена поддержка Unicode 6.2.0;

Добавлен стандартный класс для работы с Base64;

Добавлена поддержка беззнаковой арифметики;

Улучшена производительность конструктора java.lang.String(byte[], \*) и метода java.lang.String.getBytes();

Новая реализация AccessController.doPrivileged, позволяющая устанавливать подмножество привилегий без необходимости проверки всех остальных уровней доступа;

Password-based алгоритмы стали более устойчивыми;

Добавлена поддержка SSL/TLS Server Name Indication (NSI) в JSSE Server;

Улучшено хранилище ключей (KeyStore);

Добавлен алгоритм SHA-224;

Удален мост JDBC - ODBC;

Удален PermGen, изменен способ хранения мета-данных классов;

Возможность создания профилей для платформы Java SE, которые включают в себя не всю платформу целиком, а некоторую ее часть;

Инструментарий

Добавлена утилита jjs для использования JavaScript Nashorn;

Команда java может запускать JavaFX приложения;

Добавлена утилита jdeps для анализа .class-файлов.

**Что такое «лямбда»? Какова структура и особенности использования лямбда-выражения?**

Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая, собственно, представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

interface Operationable {

int calculate(int x, int y);

}

public static void main(String[] args) {

Operationable operation = (x, y) -> x + y;

int result = operation.calculate(10, 20);

System.out.println(result); //30

}

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java.

Отложенное выполнение (deferred execution) лямбда-выражения- определяется один раз в одном месте программы, вызываются при необходимости, любое количество раз и в произвольном месте программы.

Параметры лямбда-выражения должны соответствовать по типу параметрам метода функционального интерфейса:

operation = (int x, int y) -> x + y;

//При написании самого лямбда-выражения тип параметров разрешается не указывать:

(x, y) -> x + y;

//Если метод не принимает никаких параметров, то пишутся пустые скобки, например,

() -> 30 + 20;

//Если метод принимает только один параметр, то скобки можно опустить:

n -> n \* n;

Конечные лямбда-выражения не обязаны возвращать какое-либо значение.

interface Printable {

void print(String s);

}

public static void main(String[] args) {

Printable printer = s -> System.out.println(s);

printer.print("Hello, world");

}

Блочные лямбда-выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return:

Operationable operation = (int x, int y) -> {

if (y == 0) {

return 0;

}

else {

return x / y;

}

};

Передача лямбда-выражения в качестве параметра метода:

interface Condition {

boolean isAppropriate(int n);

}

private static int sum(int[] numbers, Condition condition) {

int result = 0;

for (int i : numbers) {

if (condition.isAppropriate(i)) {

result += i;

}

}

return result;

}

public static void main(String[] args) {

System.out.println(sum(new int[] {0, 1, 0, 3, 0, 5, 0, 7, 0, 9}, (n) -> n != 0));

}

**К каким переменным есть доступ у лямбда-выражений?**

Доступ к переменным внешней области действия из лямбда-выражения очень схож к доступу из анонимных объектов. Можно ссылаться на:

неизменяемые (effectively final - не обязательно помеченные как final) локальные переменные;

поля класса;

статические переменные.

К методам по умолчанию реализуемого функционального интерфейса обращаться внутри лямбда-выражения запрещено.

**Как отсортировать список строк с помощью лямбда-выражения?**

public static List<String> sort(List<String> list){

Collections.sort(list, (a, b) -> a.compareTo(b));

return list;

}

**Что такое «ссылка на метод»?**

Если существующий в классе метод уже делает все, что необходимо, то можно воспользоваться механизмом method reference (ссылка на метод) для непосредственной передачи этого метода. Такая ссылка передается в виде:

имя\_класса::имя\_статического\_метода для статического метода;

объект\_класса::имя\_метода для метода экземпляра;

название\_класса::new для конструктора.

Результат будет в точности таким же, как в случае определения лямбда-выражения, которое вызывает этот метод.

private interface Measurable {

public int length(String string);

}

public static void main(String[] args) {

Measurable a = String::length;

System.out.println(a.length("abc"));

}

Ссылки на методы потенциально более эффективны, чем использование лямбда-выражений. Кроме того, они предоставляют компилятору более качественную информацию о типе и при возможности выбора между использованием ссылки на существующий метод и использованием лямбда-выражения, следует всегда предпочитать использование ссылки на метод.

**Какие виды ссылок на методы вы знаете?**

на статический метод;

на метод экземпляра;

на конструкторе.

Объясните выражение System.out::println.

Данное выражение иллюстрирует механизм instance method reference: передачи ссылки на метод println() статического поля out класса System.

**Что такое «функциональные интерфейсы»?**

Функциональный интерфейс - это интерфейс, который определяет только один абстрактный метод.

Чтобы точно определить интерфейс как функциональный, добавлена аннотация @FunctionalInterface, работающая по принципу @Override. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

Интерфейс может включать сколько угодно default методов и при этом оставаться функциональным, потому что default методы - не абстрактные.

**Для чего нужны функциональные интерфейсы Function<T,R>, DoubleFunction<R>, IntFunction<R> и LongFunction<R>?**

Function<T, R> - интерфейс, с помощью которого реализуется функция, получающая на вход экземпляр класса T и возвращающая на выходе экземпляр класса R.

Методы по умолчанию могут использоваться для построения цепочек вызовов (compose, andThen).

Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

DoubleFunction<R> - функция, получающая на вход Double и возвращающая на выходе экземпляр класса R;

IntFunction<R> - функция, получающая на вход Integer и возвращающая на выходе экземпляр класса R;

LongFunction<R> - функция, получающая на вход Long и возвращающая на выходе экземпляр класса R.

**Для чего нужны функциональные интерфейсы UnaryOperator<T>, DoubleUnaryOperator, IntUnaryOperator и LongUnaryOperator?**

UnaryOperator<T> (унарный оператор) принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T:

UnaryOperator<Integer> operator = x -> x \* x;

System.out.println(operator.apply(5)); // 25

DoubleUnaryOperator - унарный оператор, получающий на вход Double;

IntUnaryOperator - унарный оператор, получающий на вход Integer;

LongUnaryOperator - унарный оператор, получающий на вход Long.

**Для чего нужны функциональные интерфейсы BinaryOperator<T>, DoubleBinaryOperator, IntBinaryOperator и LongBinaryOperator?**

BinaryOperator<T> (бинарный оператор) - интерфейс, с помощью которого реализуется функция, получающая на вход два экземпляра класса T и возвращающая на выходе экземпляр класса T.

BinaryOperator<Integer> operator = (a, b) -> a + b;

System.out.println(operator.apply(1, 2)); // 3

DoubleBinaryOperator - бинарный оператор, получающий на вход Double;

IntBinaryOperator - бинарный оператор, получающий на вход Integer;

LongBinaryOperator - бинарный оператор, получающий на вход Long.

**Для чего нужны функциональные интерфейсы Predicate<T>, DoublePredicate, IntPredicate и LongPredicate?**

Predicate<T> (предикат) - интерфейс, с помощью которого реализуется функция, получающая на вход экземпляр класса T и возвращающая на выходе значение типа boolean.

Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия (and, or, negate).

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true

predicate.negate().test("foo"); // false

DoublePredicate - предикат, получающий на вход Double;

IntPredicate - предикат, получающий на вход Integer;

LongPredicate - предикат, получающий на вход Long.

**Для чего нужны функциональные интерфейсы Consumer<T>, DoubleConsumer, IntConsumer и LongConsumer?**

Consumer<T> (потребитель) - интерфейс, с помощью которого реализуется функция, которая получает на вход экземпляр класса T, производит с ним некоторое действие и ничего не возвращает.

Consumer<String> hello = (name) -> System.out.println("Hello, " + name);

hello.accept("world");

DoubleConsumer - потребитель, получающий на вход Double;

IntConsumer - потребитель, получающий на вход Integer;

LongConsumer - потребитель, получающий на вход Long.

**Для чего нужны функциональные интерфейсы Supplier<T>, BooleanSupplier, DoubleSupplier, IntSupplier и LongSupplier?**

Supplier<T> (поставщик) - интерфейс, с помощью которого реализуется функция, ничего не принимающая на вход, но возвращающая на выход результат класса T;

Supplier<LocalDateTime> now = LocalDateTime::now;

now.get();

DoubleSupplier - поставщик, возвращающий Double;

IntSupplier - поставщик, возвращающий Integer;

LongSupplier - поставщик, возвращающий Long.

**Для чего нужен функциональный интерфейс BiConsumer<T,U>?**

BiConsumer<T,U> представляет собой операцию, которая принимает два аргумента классов T и U производит с ними некоторое действие и ничего не возвращает.

**Для чего нужен функциональный интерфейс BiFunction<T,U,R>?**

BiFunction<T,U,R> представляет собой операцию, которая принимает два аргумента классов T и U и возвращающая результат класса R.

**Для чего нужен функциональный интерфейс BiPredicate<T,U>?**

BiPredicate<T,U> представляет собой операцию, которая принимает два аргумента классов T и U и возвращающая результат типа boolean.

**Для чего нужны функциональные интерфейсы вида \_To\_Function?**

DoubleToIntFunction - операция, принимающая аргумент класса Double и возвращающая результат типа Integer;

DoubleToLongFunction - операция, принимающая аргумент класса Double и возвращающая результат типа Long;

IntToDoubleFunction - операция, принимающая аргумент класса Integer и возвращающая результат типа Double;

IntToLongFunction - операция, принимающая аргумент класса Integer и возвращающая результат типа Long;

LongToDoubleFunction - операция, принимающая аргумент класса Long и возвращающая результат типа Double;

LongToIntFunction - операция, принимающая аргумент класса Long и возвращающая результат типа Integer.

**Для чего нужны функциональные интерфейсы ToDoubleBiFunction<T,U>, ToIntBiFunction<T,U> и ToLongBiFunction<T,U>?**

ToDoubleBiFunction<T,U> - операция принимающая два аргумента классов T и U и возвращающая результат типа Double;

ToLongBiFunction<T,U> - операция принимающая два аргумента классов T и U и возвращающая результат типа Long;

ToIntBiFunction<T,U> - операция принимающая два аргумента классов T и U и возвращающая результат типа Integer.

**Для чего нужны функциональные интерфейсы ToDoubleFunction<T>, ToIntFunction<T> и ToLongFunction<T>?**

ToDoubleFunction<T> - операция, принимающая аргумент класса T и возвращающая результат типа Double;

ToLongFunction<T> - операция, принимающая аргумент класса T и возвращающая результат типа Long;

ToIntFunction<T> - операция, принимающая аргумент класса T и возвращающая результат типа Integer.

**Для чего нужны функциональные интерфейсы ObjDoubleConsumer<T>, ObjIntConsumer<T> и ObjLongConsumer<T>?**

ObjDoubleConsumer<T> - операция, которая принимает два аргумента классов T и Double, производит с ними некоторое действие и ничего не возвращает;

ObjLongConsumer<T> - операция, которая принимает два аргумента классов T и Long, производит с ними некоторое действие и ничего не возвращает;

ObjIntConsumer<T> - операция, которая принимает два аргумента классов T и Integer, производит с ними некоторое действие и ничего не возвращает.

**Что такое StringJoiner?**

Класс StringJoiner используется, чтобы создать последовательность строк, разделенных разделителем с возможностью присоединить к полученной строке префикс и суффикс:

StringJoiner joiner = new StringJoiner(".", "prefix-", "-suffix");

for (String s : "Hello the brave world".split(" ")) {

joiner.add(s);

}

System.out.println(joiner); //prefix-Hello.the.brave.world-suffix

**Что такое default методы интрефейса?**

Java 8 позволяет добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово default:

interface Example {

int process(int a);

default void show() {

System.out.println("default show()");

}

}

Если класс реализует интерфейс, он может, но не обязан, реализовать методы по-умолчанию, уже реализованные в интерфейсе. Класс наследует реализацию по умолчанию.

Если некий класс реализует несколько интерфейсов, которые имеют одинаковый метод по умолчанию, то класс должен реализовать метод с совпадающей сигнатурой самостоятельно. Ситуация аналогична, если один интерфейс имеет метод по умолчанию, а в другом этот же метод является абстрактным - никакой реализации по умолчанию классом не наследуется.

Метод по умолчанию не может переопределить метод класса java.lang.Object.

Помогают реализовывать интерфейсы без страха нарушить работу других классов.

Позволяют избежать создания служебных классов, так как все необходимые методы могут быть представлены в самих интерфейсах.

Дают свободу классам выбрать метод, который нужно переопределить.

Одной из основных причин внедрения методов по умолчанию является возможность коллекций в Java 8 использовать лямбда-выражения.

**Как вызывать default метод интерфейса в реализующем этот интерфейс классе?**

Используя ключевое слово super вместе с именем интерфейса:

interface Paper {

default void show() {

System.out.println("default show()");

}

}

class Licence implements Paper {

public void show() {

Paper.super.show();

}

}

**Что такое static метод интерфейса?**

Статические методы интерфейса похожи на методы по умолчанию, за исключением того, что для них отсутствует возможность переопределения в классах, реализующих интерфейс.

Статические методы в интерфейсе являются частью интерфейса без возможности использовать их для объектов класса реализации;

Методы класса java.lang.Object нельзя переопределить как статические;

Статические методы в интерфейсе используются для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.

**Как вызывать static метод интерфейса?**

Используя имя интерфейса:

interface Paper {

static void show() {

System.out.println("static show()");

}

}

class Licence {

public void showPaper() {

Paper.show();

}

}

**Что такое Optional?**

Опциональное значение Optional — это контейнер для объекта, который может содержать или не содержать значение null. Такая обёртка является удобным средством предотвращения NullPointerException, т.к. имеет некоторые функции высшего порядка, избавляющие от добавления повторяющихся if null/notNull проверок:

Optional<String> optional = Optional.of("hello");

optional.isPresent(); // true

optional.ifPresent(s -> System.out.println(s.length())); // 5

optional.get(); // "hello"

optional.orElse("ops..."); // "hello"

**Что такое Stream?**

Интерфейс java.util.Stream представляет собой последовательность элементов, над которой можно производить различные операции.

Операции над стримами бывают или промежуточными (intermediate) или конечными (terminal). Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же стрим. Таким образом вы можете строить цепочки из несколько операций над одним и тем же стримом.

У стрима может быть сколько угодно вызовов промежуточных операций и последним вызов конечной операции. При этом все промежуточные операции выполняются лениво и пока не будет вызвана конечная операция никаких действий на самом деле не происходит (похоже на создание объекта Thread или Runnable, без вызова start()).

Стримы создаются на основе каких-либо источников, например классов из java.util.Collection.

Ассоциативные массивы (maps), например, HashMap, не поддерживаются.

Операции над стримами могут выполняться как последовательно, так и параллельно.

Потоки не могут быть использованы повторно. Как только была вызвана какая-нибудь конечная операция, поток закрывается.

Кроме универсальных объектных существуют особые виды стримов для работы с примитивными типами данных int, long и double: IntStream, LongStream и DoubleStream. Эти примитивные стримы работают так же, как и обычные объектные, но со следующими отличиями:

используют специализированные лямбда-выражения, например, IntFunction или IntPredicate вместо Function и Predicate;

поддерживают дополнительные конечные операции sum(), average(), mapToObj().

**Какие существуют способы создания стрима?**

Из коллекции:

Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();

Из набора значений:

Stream<String> fromValues = Stream.of("x", "y", "z");

Из массива:

Stream<String> fromArray = Arrays.stream(new String[]{"x", "y", "z"});

Из файла (каждая строка в файле будет отдельным элементом в стриме):

Stream<String> fromFile = Files.lines(Paths.get("input.txt"));

Из строки:

IntStream fromString = "0123456789".chars();

С помощью Stream.builder():

Stream<String> fromBuilder = Stream.builder().add("z").add("y").add("z").build();

С помощью Stream.iterate() (бесконечный):

Stream<Integer> fromIterate = Stream.iterate(1, n -> n + 1);

С помощью Stream.generate() (бесконечный):

Stream<String> fromGenerate = Stream.generate(() -> "0");

**В чем разница между Collection и Stream?**

Коллекции позволяют работать с элементами по-отдельности, тогда как стримы так делать не позволяют, но вместо этого предоставляют возможность выполнять функции над данными как над одним целым.

Также стоит отметить важность самой концепции сущностей: Collection - это прежде всего воплощение Структуры Данных. Например, Set не просто хранит в себе элементы, он реализует идею множества с уникальными элементами, тогда как Stream, это прежде всего абстракция необходимая для реализации конвейера вычислений, собственно, поэтому, результатом работы конвейера являются те или иные Структуры Данных или же результаты проверок/поиска и т.п.

**Для чего нужен метод collect() в стримах?**

Метод collect() является конечной операцией, которая используется для представление результата в виде коллекции или какой-либо другой структуры данных.

collect() принимает на вход Collector<Тип\_источника, Тип\_аккумулятора, Тип\_результата>, который содержит четыре этапа: supplier - инициализация аккумулятора, accumulator - обработка каждого элемента, combiner - соединение двух аккумуляторов при параллельном выполнении, [finisher] - необязательный метод последней обработки аккумулятора. В Java 8 в классе Collectors реализовано несколько распространённых коллекторов:

toList(), toCollection(), toSet() - представляют стрим в виде списка, коллекции или множества;

toConcurrentMap(), toMap() - позволяют преобразовать стрим в Map;

averagingInt(), averagingDouble(), averagingLong() - возвращают среднее значение;

summingInt(), summingDouble(), summingLong() - возвращает сумму;

summarizingInt(), summarizingDouble(), summarizingLong() - возвращают SummaryStatistics с разными агрегатными значениями;

partitioningBy() - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>;

groupingBy() - разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>;

mapping() - дополнительные преобразования значений для сложных Collector-ов.

Так же существует возможность создания собственного коллектора через Collector.of():

Collector<String, List<String>, List<String>> toList = Collector.of(

ArrayList::new,

List::add,

(l1, l2) -> { l1.addAll(l2); return l1; }

);

**Для чего в стримах применяются методы forEach() и forEachOrdered()?**

forEach() применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется;

forEachOrdered() применяет функцию к каждому объекту стрима с сохранением порядка элементов.

**Для чего в стримах предназначены методы map() и mapToInt(), mapToDouble(), mapToLong()?**

Метод map() является промежуточной операцией, которая заданным образом преобразует каждый элемент стрима.

mapToInt(), mapToDouble(), mapToLong() - аналоги map(), возвращающие соответствующий числовой стрим (то есть стрим из числовых примитивов):

Stream

.of("12", "22", "4", "444", "123")

.mapToInt(Integer::parseInt)

.toArray(); //[12, 22, 4, 444, 123]

**Какова цель метода filter() в стримах?**

Метод filter() является промежуточной операцией принимающей предикат, который фильтрует все элементы, возвращая только те, что соответствуют условию.

**Для чего в стримах предназначен метод limit()?**

Метод limit() является промежуточной операцией, которая позволяет ограничить выборку определенным количеством первых элементов.

**Для чего в стримах предназначен метод sorted()?**

Метод sorted() является промежуточной операцией, которая позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator.

Порядок элементов в исходной коллекции остается нетронутым - sorted() всего лишь создает его отсортированное представление.

**Для чего в стримах предназначены методы flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong()?**

Метод flatMap() похож на map, но может создавать из одного элемента несколько. Таким образом, каждый объект будет преобразован в ноль, один или несколько других объектов, поддерживаемых потоком. Наиболее очевидный способ применения этой операции — преобразование элементов контейнера при помощи функций, которые возвращают контейнеры.

Stream

.of("H e l l o", "w o r l d !")

.flatMap((p) -> Arrays.stream(p.split(" ")))

.toArray(String[]::new);//["H", "e", "l", "l", "o", "w", "o", "r", "l", "d", "!"]

flatMapToInt(), flatMapToDouble(), flatMapToLong() - это аналоги flatMap(), возвращающие соответствующий числовой стрим.

**Расскажите о параллельной обработке в Java 8.**

Стримы могут быть последовательными и параллельными. Операции над последовательными стримами выполняются в одном потоке процессора, над параллельными — используя несколько потоков процессора. Параллельные стримы используют общий ForkJoinPool доступный через статический ForkJoinPool.commonPool() метод. При этом, если окружение не является многоядерным, то поток будет выполняться как последовательный. Фактически применение параллельных стримов сводится к тому, что данные в стримах будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются конечные операции.

Для создания параллельного потока из коллекции можно также использовать метод parallelStream() интерфейса Collection.

Чтобы сделать обычный последовательный стрим параллельным, надо вызвать у объекта Stream метод parallel(). Метод isParallel() позволяет узнать является ли стрим параллельным.

С помощью, методов parallel() и sequential() можно определять какие операции могут быть параллельными, а какие только последовательными. Так же из любого последовательного стрима можно сделать параллельный и наоборот:

collection

.stream()

.peek(...) // операция последовательна

.parallel()

.map(...) // операция может выполняться параллельно,

.sequential()

.reduce(...) // операция снова последовательна

Как правило, элементы передаются в стрим в том же порядке, в котором они определены в источнике данных. При работе с параллельными стримами система сохраняет порядок следования элементов. Исключение составляет метод forEach(), который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод forEachOrdered().

Критерии, которые могут повлиять на производительность в параллельных стримах:

Размер данных - чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.

Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.

Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из ArrayList легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных. А вот коллекция типа LinkedList - не лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.

Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов.

Крайне не рекомендуется использовать параллельные стримы для скольких-нибудь долгих операций (например, сетевых соединений), так как все параллельные стримы работают c одним ForkJoinPool, то такие долгие операции могут остановить работу всех параллельных стримов в JVM из-за отсутствия доступных потоков в пуле, т.е. параллельные стримы стоит использовать лишь для коротких операций, где счет идет на миллисекунды, но не для тех где счет может идти на секунды и минуты;

Сохранение порядка в параллельных стримах увеличивает издержки при выполнении и если порядок не важен, то имеется возможность отключить его сохранение и тем самым увеличить производительность, использовав промежуточную операцию unordered():

collection.parallelStream()

.sorted()

.unordered()

.collect(Collectors.toList());

**Какие конечные методы работы со стримами вы знаете?**

findFirst() возвращает первый элемент;

findAny() возвращает любой подходящий элемент;

collect() представление результатов в виде коллекций и других структур данных;

count() возвращает количество элементов;

anyMatch() возвращает true, если условие выполняется хотя бы для одного элемента;

noneMatch() возвращает true, если условие не выполняется ни для одного элемента;

allMatch() возвращает true, если условие выполняется для всех элементов;

min() возвращает минимальный элемент, используя в качестве условия Comparator;

max() возвращает максимальный элемент, используя в качестве условия Comparator;

forEach() применяет функцию к каждому объекту (порядок при параллельном выполнении не гарантируется);

forEachOrdered() применяет функцию к каждому объекту с сохранением порядка элементов;

toArray() возвращает массив значений;

reduce()позволяет выполнять агрегатные функции и возвращать один результат.

Для числовых стримов дополнительно доступны:

sum() возвращает сумму всех чисел;

average() возвращает среднее арифметическое всех чисел.

**Какие промежуточные методы работы со стримами вы знаете?**

filter() отфильтровывает записи, возвращая только записи, соответствующие условию;

skip() позволяет пропустить определённое количество элементов в начале;

distinct() возвращает стрим без дубликатов (для метода equals());

map() преобразует каждый элемент;

peek() возвращает тот же стрим, применяя к каждому элементу функцию;

limit() позволяет ограничить выборку определенным количеством первых элементов;

sorted() позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator;

mapToInt(), mapToDouble(), mapToLong() - аналоги map() возвращающие стрим числовых примитивов;

flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong() - похожи на map(), но могут создавать из одного элемента несколько.

Для числовых стримов дополнительно доступен метод mapToObj(), который преобразует числовой стрим обратно в объектный

**Как вывести на экран 10 случайных чисел, используя forEach()?**

(new Random())

.ints()

.limit(10)

.forEach(System.out::println);

**Как можно вывести на экран уникальные квадраты чисел используя метод map()?**

Stream

.of(1, 2, 3, 2, 1)

.map(s -> s \* s)

.distinct()

.forEach(System.out::println);

**Как вывести на экран количество пустых строк с помощью метода filter()?**

System.out.println(

Stream

.of("Hello", "", ", ", "world", "!")

.filter(String::isEmpty)

.count());

**Как вывести на экран 10 случайных чисел в порядке возрастания?**

(new Random())

.ints()

.limit(10)

.sorted()

.forEach(System.out::println);

**Как найти максимальное число в наборе?**

Stream

.of(5, 3, 4, 55, 2)

.mapToInt(a -> a)

.max()

.getAsInt(); //55

**Как найти минимальное число в наборе?**

Stream

.of(5, 3, 4, 55, 2)

.mapToInt(a -> a)

.min()

.getAsInt(); //2

**Как получить сумму всех чисел в наборе?**

Stream

.of(5, 3, 4, 55, 2)

.mapToInt()

.sum(); //69

**Как получить среднее значение всех чисел?**

Stream

.of(5, 3, 4, 55, 2)

.mapToInt(a -> a)

.average()

.getAsDouble(); //13.8

**Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8?**

putIfAbsent() добавляет пару «ключ-значение», только если ключ отсутствовал:

map.putIfAbsent("a", "Aa");

forEach() принимает функцию, которая производит операцию над каждым элементом:

map.forEach((k, v) -> System.out.println(v));

compute() создаёт или обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

map.compute("a", (k, v) -> String.valueOf(k).concat(v)); //["a", "aAa"]

computeIfPresent() если ключ существует, обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

map.computeIfPresent("a", (k, v) -> k.concat(v));

computeIfAbsent() если ключ отсутствует, создаёт его со значением, которое вычисляется (возможно использовать ключ):

map.computeIfAbsent("a", k -> "A".concat(k)); //["a","Aa"]

getOrDefault() в случае отсутствия ключа, возвращает переданное значение по-умолчанию:

map.getOrDefault("a", "not found");

merge() принимает ключ, значение и функцию, которая объединяет передаваемое и текущее значения. Если под заданным ключем значение отсутствует, то записывает туда передаваемое значение.

map.merge("a", "z", (value, newValue) -> value.concat(newValue)); //["a","Aaz"]

**Что такое LocalDateTime?**

LocalDateTime объединяет вместе LocaleDate и LocalTime, содержит дату и время в календарной системе ISO-8601 без привязки к часовому поясу. Время хранится с точностью до наносекунды. Содержит множество удобных методов, таких как plusMinutes, plusHours, isAfter, toSecondOfDay и т.д.

**Что такое ZonedDateTime?**

java.time.ZonedDateTime — аналог java.util.Calendar, класс с самым полным объемом информации о временном контексте в календарной системе ISO-8601. Включает временную зону, поэтому все операции с временными сдвигами этот класс проводит с её учётом.

**Как получить текущую дату с использованием Date Time API из Java 8?**

LocalDate.now();

**Как добавить 1 неделю, 1 месяц, 1 год, 10 лет к текущей дате с использованием Date Time API?**

LocalDate.now().plusWeeks(1);

LocalDate.now().plusMonths(1);

LocalDate.now().plusYears(1);

LocalDate.now().plus(1, ChronoUnit.DECADES);

**Как получить следующий вторник используя Date Time API?**

LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.TUESDAY));

**Как получить вторую субботу текущего месяца используя Date Time API?**

LocalDate

.of(LocalDate.now().getYear(), LocalDate.now().getMonth(), 1)

.with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY))

.with(TemporalAdjusters.next(DayOfWeek.SATURDAY));

**Как получить текущее время с точностью до миллисекунд используя Date Time API?**

new Date().toInstant();

**Как получить текущее время по местному времени с точностью до миллисекунд используя Date Time API?**

LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault());

**Как определить повторяемую аннотацию?**

Чтобы определить повторяемую аннотацию, необходимо создать аннотацию-контейнер для списка повторяемых аннотаций и обозначить повторяемую мета-аннотацией @Repeatable:

@interface Schedulers

{

Scheduler[] value();

}

@Repeatable(Schedulers.class)

@interface Scheduler

{

String birthday() default "Jan 8 1935";

}