Оглавление

Core-1

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.

Какие преимущества у ООП?

Легко читается – не нужно выискивать в коде функции и выяснять, за что они отвечают.

Быстро пишется – можно быстро создать сущности, с которыми должна работать программа.

Простота реализации большого функционала – т. к. на написание кода уходит меньше времени, можно гораздо быстрее создать приложение с множеством возможностей.

Какие недостатки у ООП?

Потребление памяти – объекты потребляют больше оперативной памяти, чем примитивные типы данных.

Снижается производительность – многие вещи технически реализованы иначе, поэтому они используют больше ресурсов.

Сложно начать – парадигма ООП сложнее функционального программирования, поэтому на старт уходит больше времени.

Принципы ООП (наследование, инкапсуляция, полиморфизм, абстракция)

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.

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.

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).

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

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, четко определяя все возможные действия над ним.

Ассоциация, агрегация, композиция

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.

Является – «is a», имеет – «has a»

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

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).

SOLID

Принцип единственной ответственности

Класс должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

Принцип открытости-закрытости

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

Принцип подстановки Барбары Лисков

Необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов.

Цель этого принципа заключаются в том, чтобы классы-наследники могли бы использоваться вместо родительских классов, от которых они образованы, не нарушая работу программы. Если оказывается, что в коде проверяется тип класса, значит принцип подстановки нарушается.

Принцип разделения интерфейса

Создание узкоспециализированных интерфейсов, предназначенных для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.

Этот принцип направлен на устранение недостатков, связанных с реализацией больших интерфейсов.

Принцип инверсии зависимостей

Объектом зависимости должна быть абстракция, а не что-то конкретное.

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

В процессе разработки программного обеспечения существует момент, когда функционал приложения перестает помещаться в рамках одного модуля. Когда это происходит, приходится решать проблему зависимостей модулей. В результате, например, может оказаться так, что высокоуровневые компоненты зависят от низкоуровневых компонентов.

https://ota-solid.now.sh/lsp

Какая основная идея языка?

«Написано однажды – работает везде» (**WORA**).

Идея основывается в написании одного кода, который будет работать на любой платформе.

За счет чего обеспечивается кроссплатформенность?

Кроссплатформенность была достигнута **за счет** создания **виртуальной машины Java**. Java Virtual Machine или JVM – это программа, являющаяся **прослойкой между операционной системой** и **Java-программой**. В среде виртуальной машины выполняются коды Java-программ. Сама JVM реализована для разных ОС.

Какие преимущества у Java?

Объектно-ориентированное программирование – структура данных становится объектом, которым можно управлять для создания отношений между различными объектами.

Язык высокого уровня с простым синтаксисом и плавной кривой обучения – синтаксис Java основан на C ++, поэтому Java похожа на C. Тем не менее, синтаксис Java проще, что позволяет новичкам быстрее учиться и эффективнее использовать код для достижения конкретных результатов.

Стандарт для корпоративных вычислительных систем – корпоративные приложения – главное преимущество Java с 90-х годов, когда организации начали искать надежные инструменты программирования не на C.

Безопасность – благодаря отсутствию указателей и Security Manager (политика безопасности, в которой можно указать правила доступа, позволяет запускать приложения Java в «песочнице»).

Независимость от платформы – можно создать Java-приложение на Windows, скомпилировать его в байт-код и запустить его на любой другой платформе, поддерживающей виртуальную машину Java (JVM). Таким образом, JVM служит уровнем абстракции между кодом и оборудованием.

GC – garbage collector (сборщик мусора).

Язык для распределенного программирования и комфортной удаленной совместной работы – специфическая для Java методология распределенных вычислений называется Remote Method Invocation (RMI). RMI позволяет использовать все преимущества Java: безопасность, независимость от платформы и объектно-ориентированное программирование для распределенных вычислений. Кроме того, Java также поддерживает программирование сокетов и методологию распределения CORBA для обмена объектами между программами, написанными на разных языках.

Автоматическое управление памятью – разработчикам Java не нужно вручную писать код для управления памятью благодаря автоматическому управлению памятью (AMM).

Многопоточность – поток – наименьшая единица обработки в программировании. Чтобы максимально эффективно использовать время процессора, Java позволяет запускать потоки одновременно, что называется многопоточностью.

Стабильность и сообщество – сообщество разработчиков Java не имеет себе равных. Около 45% респондентов опроса StackOverflow (2018) используют Java.

Какие недостатки у java?

- **платное коммерческое использование** (с 2019);
- **низкая производительность** из-за компиляции и абстракции с помощью виртуальной машины, а также приложения очистки памяти (из-за кроссплатформенности, GC, обратной совместимости, скорости развертывания);
- **не развитые инструменты по созданию GUI-приложений** на чистой java.
- **многословный код** – Java – это более легкая версия неприступного C++, которая вынуждает программистов прописывать свои действия словами из английского языка, это делает язык более понятным для неспециалистов, но менее компактным.

Что такое JDK? Что в него входит?

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.

Что такое JRE? Что в него входит?

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.

Что такое JVM?

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.

Что такое byte code?

Байт-код Java – набор инструкций, скомпилированный компилятором, исполняемый JVM, имеется уже около 200 инструкций, 56 в запасе. 1 инструкция = 1 байту.

Что такое загрузчик классов (classloader)?

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, 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 неизвестны.
- **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 свое пространство имен 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 JIT?

JIT-компиляция (англ. Just-in-time compilation, компиляция «на лету»), динамическая компиляция (англ. dynamic translation) – технология увеличения производительности программных систем, использующих байт-код, путем компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы.

Виды ссылок в Java

В Java существует 4 типа ссылок. Особенности каждого типа ссылок связаны с работой Garbage Collector.

- сильные (**strong reference**);
- мягкие (**SoftReference**);
- слабые (**WeakReference**);
- фантомные (**PhantomReference**).

Отличия между слабыми, мягкими, фантомными и обычными ссылками в Java

«Слабые» ссылки и «мягкие» ссылки (**WeakReference**, **SoftReference**) были добавлены в Java API давно. Ссылочные классы особенно важны в контексте сборки мусора. Сборщик мусора сам освобождает память, занимаемую объектами, но решение об освобождении памяти он принимает, исходя из типа имеющихся на объект ссылок.

Главное отличие *SoftReference* от *WeakReference* в том, как сборщик с ними будет работать. Он может удалить объект в любой момент, если на него указывают только weak-ссылки, с другой стороны объекты с soft-ссылкой будут собраны только когда JVM очень нужна память. Благодаря таким особенностям ссылочных классов каждый из них имеет свое применение. SoftReference можно использовать для реализации кэшей, и когда JVM понадобится память, она освободит ее за счет удаления таких объектов. А WeakReference отлично подойдут для

хранения метаданных, например, для хранения ссылки на ClassLoader. Если нет классов для загрузки, то нет смысла хранить ссылку на ClassLoader, слабая ссылка делает ClassLoader доступным для удаления как только мы назначим ее вместо сильной ссылки (Strong reference).

Фантомные ссылки – третий тип ссылок, доступных в пакете java.lang.ref. Phantom- ссылки представлены классом java.lang.ref.PhantomReference. Объект, на который указывают только phantom-ссылки, может быть удален сборщиком в любой момент. Phantom-ссылка создается точно так же, как weak или soft.

```
DigitalCounter digit = new DigitalCounter(); // digit reference variable has strong reference

PhantomReference phantom = new PhantomReference(digit); // phantom reference

digit = null;
```

Как только обнулите strong-ссылки на объект DigitalCounter, сборщик мусора удалит его в любой момент, так как теперь на него ведут только phantom-ссылки.

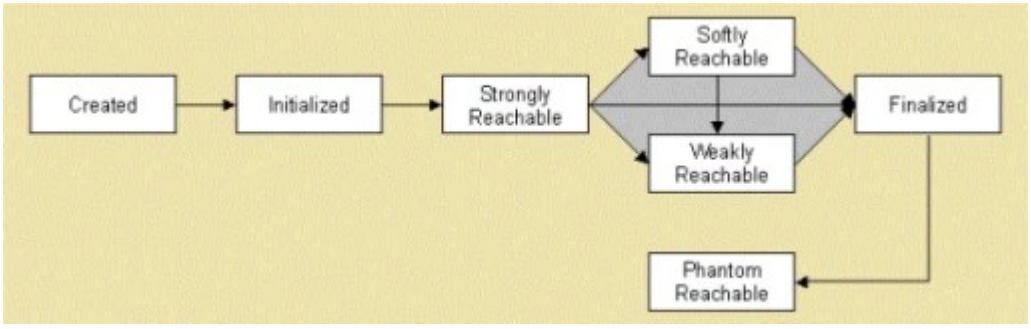
Классом **ReferenceQueue** можно воспользоваться при создании объекта класса WeakReference, SoftReference или PhantomReference:

```
ReferenceQueue refQueue = new ReferenceQueue(); //reference will be stored in this queue for cleanup

DigitalCounter digit = new DigitalCounter();

PhantomReference phantom = new PhantomReference(digit, refQueue);
```

Ссылка на объект будет добавлена в ReferenceQueue, и можно будет контролировать состояние ссылок путем опроса ReferenceQueue. Жизненный цикл Object хорошо представлен на диаграмме:



Правильное использование ссылок поможет при сборке мусора, и в результате получим более гибкое управление памятью в Java.

https://javarush.ru/groups/posts/1267-otlichija-mezhdu-slabihmi-mjagkimi-fantomnihmi-i-obihchnihmi-ssihlkami-v-java

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 (подсчет 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 можно добраться из корневых точек (GC Root) и те объекты, которые доступны с живого объекта. Все остальное – мусор.

Существует 4 типа корневых точек:

- локальные переменные и параметры методов;
- потоки;
- статические переменные;
- ссылки из JNI.

Самое простое java-приложение будет иметь корневые точки:

- локальные переменные внутри метода main() и параметры метода main();
- поток, который выполняет main();
- статические переменные класса, внутри которого находится метод main().

Таким образом, если представим все объекты и ссылки между ними как дерево, то нужно будет пройти с корневых узлов (точек) по всем ребрам. При этом узлы, до которых 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 подходе **copying collectors** 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 неэффективно.

Сборщики мусора HotSpot VM используют комбинированный подход Generational Garbage Collection, который позволяет использовать разные алгоритмы для разных этапов сборки мусора. Этот подход основывается 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 с помощью оператора **new**, тем самым присваивая объекту ссылку. Для окончания работы с объектом достаточно 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 **-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 приложение приостанавливается и продолжает работать только после прекращение 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 трех 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 еще не достигли порога «старости» (tenuring threshold), чтобы быть перемещенными в Old generation.

Когда Young generation заполняется, то в этой области запускается процесс легкой 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 легкой 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 приложений. 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 трех фаз. В фазе **Mark (пометка)** сборщик мусора помечает все живые объекты, затем, в фазе **Sweep (очистка)** все не помеченные объекты удаляются, а в фазе **Compact (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 Мб кучи, работающих на машинах клиентского типа и не предъявляющих жестких 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 объемов 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 finalize()? Зачем он нужен?

Через вызов метода **finalize()** JVM реализуется функциональность, аналогичная функциональности деструкторов в C++, 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.

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 создается 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 к 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 свое значение внутри метода.

Оператор **finally** гарантирует, что определенный в нем 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.

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. Она 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 находится 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 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 из 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 опции JVM **-Xms** и **-Xmx**. 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 (все, что явно или неявно создается при помощи оператора **new**) хранится в куче.

Ключевые слова

abstract, assert, break, case, catch, class, const*, continue, default, do, else, enum, extends, final, finally, for, goto*, if, implements, import, instanceof, interface, native, new, package, return, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while.

** – зарезервированное слово, не используется.*

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 проверка 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 задает детальное сообщение о произошедшей ошибке (вычисленное выражение будет преобразовано в строку и передано конструктору **AssertionError**).

Какие примитивные типы данных есть в Java?

Числа инициализируются 0 или 0.0;

char – \u0000;

boolean – false;

Объекты (в том числе String) – null.

Type	Default Value	Объем памяти	Допустимые значения
byte	0	1байт	-от -128 до 127
short	0	2байта	от -32768 до 32767
int	0	4байта	от -2147483648 до 2147483647
long	0L	8байт	от -9223372036854775808L до 9223372036854775807L
float	0.0f	4байта	от 1.4e-45f до 3.4e+38f
double	0.0d	8байт	от 4.9e-324 до 1.7e+308
char	'\u0000'	2байта	от 0 до 65536
boolean	false	1байт в массиве, 4байта	true (истина) или false (ложь)

Что такое char?

16-разрядное беззнаковое целое, представляющее собой символ UTF-16 (буквы и цифры).

Сколько памяти занимает boolean?

Зависит от реализации JVM: минимум 1 байт в массивах, 4 байта в коде.

Логические операторы

&: Логическое AND (И);

&&: Сокращенное AND;

|: Логическое OR (ИЛИ);

||: Сокращенное 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.

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 учета знака;

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

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

<<=: Сдвиг влево с присваиванием.

Что такое классы-обертки?

Обертка – это специальный класс, который хранит внутри себя значение примитива. Нужны для реализации дженериков (или коллекций), объектов.

Что такое автоупаковка и автораспаковка?

Автоупаковка – присвоение классу обертки значения примитивного типа.

Автораспаковка – присвоение переменной примитивного типа значение класса обертки.

Необходимы для присваивания ссылок-примитивов объектам их классов-оберток (и наоборот). Не требуется ничего делать, все происходит автоматически.

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 кешируются JVM. Поэтому такие обертки с одинаковыми значениями будут являться ссылками на один объект.

Что такое явное и неявное приведение типов? В каких случаях в java нужно использовать явное приведение?

Java является строго типизированным языком программирования, а это означает, 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 емкому. 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 пул интов?

В классе-обертке Integer есть внутренний класс **IntegerCache** – пул (pool) целых чисел в промежутке [-128; 127], так как это самый часто встречающийся диапазон. Он объявлен как private static. В этом внутреннем классе кешированные объекты находятся в массиве cache[]. Кеширование выполняется при первом использовании класса-обертки. После первого использования вместо создания нового экземпляра (кроме использования конструктора), используются кешированные объекты, JVM берет их из пула.

Можно ли изменить размер пула int?

Не из кода, а в параметре JVM.

Какие еще есть пулы примитивов?

У всех целочисленных и char, но **размеры изменять нельзя,** можно только у **int**.

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**.

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;
- если 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 нее;
- при использовании оператора **new** создается 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).**

Почему не рекомендуется изменять строки в цикле? Что рекомендуется использовать?

Строка – неизменяемый класс, поэтому растет потребление ресурсов при редактировании, т. к. **при каждой итерации будет создаваться новый объект** строки. **Рекомендуется** использовать **StringBuilder**.

Почему char[] предпочтительнее String для хранения пароля?

С момента создания строка остается 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 определенный 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 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, ее 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, ее hashCode() кешируется 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 строка является популярным ключом в HashMap в Java?

Поскольку строки неизменяемы, их хеш-код вычисляется и кешируется 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 байт-код 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, он во всем идентичен классу StringBuffer за исключением того, что он не синхронизирован и поэтому его методы выполняются **значительно быстрей**.

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
```

Существуют ли в Java многомерные массивы?

Да (спорно). Тип данных массива – ссылочный. Массив подразумевает непрерывное хранение в памяти, все вложенные массивы будут одинаковыми, под них выделена одинаковая память, это структура, под которую выделяется объем памяти, поэтому нужно знать заранее, какой объем будет у массива.

Какими значениями инициируются переменные по умолчанию?

- byte, short, int – 0;
- long – 0L;
- float – 0.0f;
- double – 0.0d;
- char – '\u0000' (символ конца строки);
- boolean – false (зависит от реализации, можно установить true по умолчанию);
- объекты – null (это ссылка никуда не указывает, спецуказатель);
- локальные (в методе) переменные **не имеют значений по умолчанию**, их имеют поля класса;
- не static-поле класса будет инициализировано после того, как будет создан объект этого класса, а static-поле будет инициализировано тогда, когда класс будет загружен JVM;
- сколько весит ссылка в Java: на 32 бит – 4 байта, на 64 бит – 8 байт (но вроде как есть 4 байта) (железо + JVM);
- заголовок объекта – 1 бит.

Что такое сигнатура метода?

Это **имя метода** плюс **параметры** (порядок параметров имеет значение из-за множественной передачи данных через троеточие, которое должно располагаться последним). В сигнатуру метода не входит возвращаемое значение, а также бросаемые им исключения.

Сигнатура метода в сочетании с типом возвращаемого значения и бросаемыми исключениями называется **контрактом метода.**

От модификатора до выбрасываемого исключения – это **контракт**.

Расскажите про метод main

Является, как правило, **точкой входа** в программу и **вызывается JVM**.

Как только **заканчивается выполнение** метода **main()**, так сразу же **завершается работа** самой **программы**.

static – чтобы JVM смогла загрузить его во время компиляции.

public static void и сигнатура – обязательное декларирование.

Мэйнов может быть много и может не быть вообще.

Может быть перегружен.

Каким образом переменные передаются 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.

Массив – это объект.

Если передать массив и изменить его в методе, то будет ли изменяться текущий массив?

Да, текущий массив изменится тоже.

Какие типы классов есть 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.

Вложенные классы делятся на две категории: **статические** и **нестатические**. Объявленные вложенные классы static называются **статическими вложенными классами**. Нестатические вложенные классы называются **внутренними классами**.

Внутренние классы ассоциируются не с внешним классом, а с экземпляром внешнего.

1. **Статические вложенные классы** (Static nested classes)
   - есть возможность обращения к внутренним статическим полям и методам класса-обертки;
   - обрамляющий класс не имеет доступа к статическим полям;

- из статического вложенного класса не имеем доступ к статическим полям внешнего класса.

2. **Вложенные классы** бывают двух видов (Inner или Non-static Nested)

- есть возможность обращения к внутренним полям и методам класса-обертки;
- не может иметь статических объявлений;
- внутри такого класса нельзя объявить перечисления;
- если нужно явно получить this внешнего класса – OuterClass.this.

3. **Локальные классы**

- видны только в пределах блока, в котором объявлены;
- не могут быть объявлены как private/public/protected или static (по этой причине интерфейсы нельзя объявить локально);
- не могут иметь внутри себя статических объявлений (полей, методов, классов), но могут иметь константы (static final);
- имеют доступ к полям и методам обрамляющего класса;
- можно обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final или являются effectively final.

4. **Анонимные классы**

- локальный класс без имени;
- создается, чтобы его сразу же применить.

Вложенный класс – это итератор внутри коллекции.

Если связь между объектом внутреннего класса и объектом внешнего класса не нужна, можно сделать внутренний класс статическим (static). Такой класс называют вложенным (nested).

Применение статического внутреннего класса означает следующее:

- для создания объекта статического внутреннего класса не нужен объект внешнего класса;
- из объекта вложенного класса нельзя обращаться к нестатическим членам внешнего класса.

Каждый тип класса имеет рекомендации по своему применению:

- **нестатический**: 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;
- **статический**: если ссылка на обрамляющий класс не требуется;
- **локальный**: если класс необходим только внутри какого-то метода и требуется создавать экземпляры этого класса только в этом методе;
- **анонимный**: если к тому же применение класса сводится к использованию лишь в одном месте и уже существует тип, характеризующий этот класс.

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 статического 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 (полей, методов, классов);
- 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:

- 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.

Что такое перечисления (enum)?

Перечисления представляют **набор логически связанных констант**.

Перечисление фактически представляет новый класс, поэтому можно определить переменную данного типа и использовать ее.

Перечисления, как и обычные классы, **могут определять конструкторы, поля и методы.** При этом **конструктор по умолчанию приватный.** Также можно определять методы для отдельных констант.

Можно создавать публичные геттеры\сеттеры. Они создаются в момент компиляции.

Методы:

- **valueOf()** возвращает конкретный элемент;
- **ordinal()** возвращает порядковый **номер определенной константы** (нумерация начинается с 0);
- **values()** возвращает **массив всех констант** перечисления;
- **name()** отличается от toString тем, что второй можно переопределить.

В Енам реализация equals() через ==, поэтому можно и через equals(), и через ==.

Enum имеет ряд преимуществ при использовании в сравнении с static final int.

Главным отличием является то, что, **используя enum, можно проверить тип данных**.

Недостатки:

- не применимы операторы >, <, >=, <=;

- требует больше памяти для хранения, чем обычная константа.

Нужны для ограничения области допустимых значений: например, времена года, дни недели.

Особенности Enum-классов

- Конструктор всегда private или default.
- Могут имплементировать интерфейсы.
- Не могут наследовать класс.
- Можно переопределить toString().
- Нет public конструктора, поэтому нельзя создать экземпляр вне Enum.
- При equals() выполняется ==.
- ordinal() возвращает порядок элементов.
- Может использоваться в TreeSet и TreeMap, т. к. Enum имплементирует Comparable.
- compareTo() имитирует порядок элементов, предоставляемый ordinal().
- Можно использовать в Switch Case.
- values() возвращает массив всех констант.
- Легко создать потокобезопасный singleton без double check volatile переменных.

Ромбовидное наследование

Ромбовидное наследование (англ. diamond inheritance) – ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. При этой схеме наследования может возникнуть неоднозначность: если объект класса D вызывает метод, определенный в классе A (и этот метод не был переопределен в классе D), а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C?

Как проблема ромбовидного наследования решена в java?

В Java нет поддержки множественного наследования классов.

Предположим, что SuperClass – это абстрактный класс, описывающий некоторый метод, а классы ClassA и ClassB – обычные классы наследники SuperClass, а класс ClassC наследуется от ClassA и ClassB одновременно. Вызов метода родительского класса приведет к неопределенности, так как компилятор не знает о том, метод какого именно суперкласса должен быть вызван. Это и есть основная причина, почему в Java нет поддержки множественного наследования классов. Интерфейсы – это только резервирование/описание метода, а реализация самого метода будет в конкретном классе, реализующем эти интерфейсы, таким образом исключается неопределенность при множественном наследовании интерфейсов. В случае, если вызывается default-метод из интерфейса его обязательно надо будет переопределить.

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».

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

В классе-наследнике при отсутствии переопределенного конструктора будет использован конструктор родителя.

Могут ли быть приватные конструкторы? Для чего они нужны?

Да, могут. Приватный конструктор **запрещает создание экземпляра класса вне методов самого класса.**

Финальные нет.

Нужен для реализации паттернов, например, **singleton**.

Приватный конструктор запрещает вызывать конструктор другим классам извне.

У абстрактного класса есть приватный конструктор (абстрактный класс позволяет описать некоторое состояние объекта).

Расскажите про классы-загрузчики и про динамическую загрузку классов

При запуске JVM для загрузки приложения используются следующие загрузчики классов:

- **Bootstrap ClassLoader** – главный загрузчик (загружает платформенные классы JDK из архива rt.jar);
- **AppClassLoader** – системный загрузчик (загружает классы приложения текущего, определенные в CLASSPATH);
- **SystemClassLoader** загружает классы приложения, определенные в CLASSPATH;
- **Extension ClassLoader** – загрузчик расширений загружает все необходимые библиотеки из директории java.home (загружает классы расширений из javahome, которые по умолчанию находятся в каталоге jre/lib/ext).

Исключение – **ClassNotFoundExtension**.

Динамическая загрузка происходит «на лету» **в ходе выполнения программы** с помощью статического метода класса **Class.forName** (имя класса). Для чего нужна динамическая загрузка? Например, не знаем, какой класс понадобится и принимаем решение в ходе выполнения программы, передавая имя класса в статический метод forName().

Чем отличаются конструкторы 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).

Какие модификаторы доступа есть в Java? Какие применимы к классам?

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 (защищенный): 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).

Класс может быть объявлен с модификатором **public** и **default**.

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 методы, которые предоставлены разработчиком класса. Например: **getX()** и **setX()**.
- через механизм рефлексии (**Reflection API**).

Что означает модификатор static?

Статическая переменная – это переменная, принадлежащая классу, а не объекту.

Статический класс – это вложенный класс, который может обращаться только к статическим полям обертывающего его класса.

Внутри **статического метода** нельзя вызвать нестатический метод по имени класса. Можно обратиться к статическому методу через экземпляр класса.

К каким конструкциям Java применим модификатор static?

- полям;
- методам;
- вложенным классам;

- членам секции import;
- блокам инициализации.

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:

- могут вызывать только другие статические методы;
- должны осуществлять доступ только к статическим переменным;
- не могут ссылаться на члены типа **this** или **super**.

В отличие 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.

Может ли статический метод быть переопределен или перегружен?

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

Переопределен – 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.

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 сузить уровень доступа/тип возвращаемого значения при переопределении метода?

При переопределении метода **нельзя сузить модификатор доступа к методу** (например, с public до private), но можно расширить.

Изменить тип возвращаемого значения нельзя, но **можно сузить** возвращаемое значение, **если они совместимы**. Например, если метод возвращает объект класса, а переопределенный метод возвращает класс-наследник.

Что можно изменить в сигнатуре метода при переопределении? Можно ли менять модификаторы (throws и т. п.)?

При переопределении метода сужать модификатор доступа не разрешается, т. к. это приведет 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:

Поэтому **в сигнатуре** (имя + параметры) менять ничего **нельзя**, но **возможно расширение уровня доступа**.

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

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

Могут ли классы быть статическими?

Класс можно объявить статическим за исключением классов верхнего уровня.

Такие классы известны как «вложенные статические классы» (nested static class).

Что означает модификатор final? К чему он может быть применим?

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 свое значение внутри метода;
- для локальных **переменных примитивного типа** это означает, что **однажды присвоенное значение не может быть изменено**;
- для **ссылочных переменных** это означает, что после присвоения объекта нельзя изменить ссылку на данный объект (**ссылку изменить нельзя, но состояние объекта изменять можно).**

Следует также отметить, что к **abstract-классам нельзя применить модификатор final**, т. к. это взаимоисключающие понятия.

Что такое абстрактные классы? Чем они отличаются от обычных?

Это обычный класс, но **с абстрактными методами**.

Особенности абстрактных классов:

- может быть конструктор (удобен для паттерна декоратор, для вызовов по цепочке из наследников);

- может содержать абстрактные методы и обычные методы;
- может создать приватный конструктор, но нужен тогда еще один, чтобы можно было его ексдендить;
- имплементят интерфейсы, но не обязаны реализовывать их методы;
- **не может быть final**, так как наследниками абстрактного класса могут быть другие абстрактные классы;
- **могут быть статические методы** (т. к. помимо наследования и переопределения абстрактный класс может использоваться без наследования);
- **нельзя создать** объект или **экземпляр абстрактного класса**;
- абстрактные методы могут отсутствовать;
- меняет хотя бы один абстрактный метод;
- абстрактный метод не может быть вне абстрактного класса;
- может содержать метод main().

Например, если много разных абстрактных классов и нужно подсчитать их, то для этого можно использовать статический метод для инкрементации (подсчет всех методов).

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.

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.

Может ли быть абстрактный класс без абстрактных методов?

Класс **может** быть абстрактным без единого абстрактного метода, если у него указан модификатор **abstract**.

Могут ли быть конструкторы у абстрактных классов? Для чего они нужны?

Да. Необходимы для наследников.

В абстрактном классе можно объявить и определить конструкторы. Даже если не объявили никакого конструктора, компилятор добавит в абстрактный класс конструктор по умолчанию

без аргументов. Абстрактные конструкторы будут часто использоваться для обеспечения ограничений класса или инвариантов, таких как минимальные поля, необходимые для настройки класса.

Что такое интерфейсы? Какие модификаторы по умолчанию имеют поля и методы интерфейсов?

Интерфейс – это совокупность методов, определяющих правила взаимодействия элементов системы. Другими словами, **интерфейс определяет как элементы будут взаимодействовать между собой**.

Ключевое слово **interface** используется для создания полностью абстрактных классов. Основное предназначение интерфейса – определять, каким образом можно 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**.

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

Интерфейс нужен чтобы реализовать абстрактный класс. По сути это абстрактный класс, все методы у него абстрактные.

- **методы интерфейса** являются публичными (**public**) и абстрактными (**abstract**), если имплементироть интерфейс, то наследующий его класс должен будет реализовать все эти абстрактные методы, в отличии от абстрактного класса;
- **поля – public static final**;
- есть дефолтный метод;
- нужно обязательно прописывать **static**;
- методы могут быть static.

После 8-й Java появились дефолтные методы – если много классов реализуют данный интерфейс и чтобы не переписывать новый метод, используется дефолтный. Т. е., чтобы избежать ромбовидное наследование, нужно переопределить этот метод. А если в одном есть дефолтный метод, а в другом нет дефолтного и нужно имплементиться от обоих, то нужно всегда переопределять дефолтный.

Чем интерфейсы отличаются от абстрактных классов? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?

1. **Интерфейс описывает** только **поведение** (методы) объекта, а вот **состояний** (полей) у него **нет** (кроме public static final), в то время как у абстрактного класса они могут быть.

2. **Можно наследовать** только **один класс**, **а реализовать интерфейсов сколько угодно**. Интерфейс может наследовать (extends) другой интерфейс/интерфейсы.

3. **Абстрактные классы используются**, когда **есть** отношение «**is-a»**, то есть класс-наследник расширяет базовый абстрактный класс, а **интерфейсы могут быть реализованы разными классами**, вовсе **не связанными друг с другом**.

4. **Абстрактный класс может реализовывать методы**; **интерфейс** может реализовывать статические методы **начиная с 8-й версии**.

5. Нет конструктора у интерфейса.

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.

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

Интерфейс.

Может ли один интерфейс наследоваться от другого? От двух других?

Да, может. Используется ключевое слово **extends**.

Что такое дефолтные методы интерфейсов? Для чего они нужны?

В JDK 8 была добавлена такая функциональность, как методы по умолчанию с модификатором default. И теперь интерфейсы могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Это нужно **для обратной совместимости**.

Если один или несколько методов добавляются к интерфейсу, все реализации также будут вынуждены их реализовывать. Методы интерфейса по умолчанию являются эффективным способом решения этой проблемы.

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.

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:

Paper.show();

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.

Как решается проблема ромбовидного наследования при наследовании интерфейсов при наличии default-методов?

Обязательным **переопределением** default-метода.

В случае, если вызывается default-метод из интерфейса, его обязательно надо будет переопределить.

Каков порядок вызова конструкторов инициализации с учетом 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.

При создании объекта производного **класса конструкторы** вызываются в **порядке** вниз по **иерархии** наследования **классов**, т. е. начиная с самого базового **класса** и заканчивая производным **классом**.

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.

Используются **для выполнения кода**, который должен выполняться один раз **при инициализации класса**.

Для чего 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 принадлежит только самому классу.

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.

Что произойдет, 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 (в контракте конструктора). Иначе будет ошибка 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.

Для **статического** блока выбрасывание исключения в явном виде приводит к ошибке компиляции **ExceptionInizialazionError**.

Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?

Если возникшее исключение – наследник **Error**:

- 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**.

Если исключение j**ava.lang.ThreadDeath** (смерть потока), то в этом случае никакое исключение выброшено не будет.

Что такое класс Object?

Все классы являются наследниками суперкласса **Object**. Это не нужно указывать явно. В результате объект **Object может ссылаться на объект любого** другого **класса**.

Какие методы есть у класса Object (перечислить все)? Что они делают?

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 определенное 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 определенное 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.

Расскажите про equals и hashcode

Хеш-код – это **целочисленный результат** работы **метода**, **которому** в качестве входного параметра **передан объект. Рассчитывается по нативному методу.**

Equals – это метод, определенный в Object, который служит для сравнения объектов. **При сравнении объектов при помощи ==** идет **сравнение по ссылкам**. **При сравнении по equals**() идет **сравнение по состояниям** объектов (по умолчанию случайным образом, но есть другие варианты).

По умолчанию ссылки, чтобы его использовать, нужно переопределить (на область в памяти), т. к. при == сравниваются ссылки, а equals сравнивает состояния:

Свойства equals():

- **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) вернет true;
- **Непротиворечивость**: для любых ссылок на значения x и y, если несколько раз вызвать x.equals(y), постоянно будет возвращаться значение true либо постоянно будет возвращаться значение false при условии, что никакая информация, используемая при сравнении объектов, не поменялась;
- **Совместимость с hashCode**(): два тождественно равных объекта должны иметь одно и то же значение hashCode().

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

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

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

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

2. 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();
```

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

Метод equals() определяет отношение эквивалентности объектов.

При сравнение объектов с помощью == сравнение происходит лишь между ссылками.

При сравнении по переопределенному разработчиком equals() – по внутреннему состоянию объектов.

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 оператора **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.

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 нее помещена, при использовании нового экземпляра ключа не будет в ней найдена.

Какой контракт между hashCode() и equals()?

1. Если два объекта возвращают разные значения hashcode(), то они не могут быть равны.
2. Если equals объектов true, то и хеш-коды должны быть равны.

3. Переопределив equals, всегда переопределять и hashcode.

Для чего нужен метод hashCode()?

Метод hashCode() необходим для вычисления хеш-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 хеш-код всегда будет одинаковым. Следует понимать, что в Java множество возможных хеш-кодов ограничено типом int, а множество объектов ничем не ограничено. Из-за этого вполне возможна ситуация, что хеш-коды разных объектов могут совпасть:

- если хеш-коды разные, то и объекты гарантированно разные;
- если хеш-коды равны, то объекты могут не обязательно равны.

В случае, если hashCode() не переопределен, то будет выполнятся его реализация по умолчанию из класса Object: для разных объектов будет разный хеш-код.

Значение int может быть в диапазоне 2^{32}, при переопределении хеш-кода можно использовать отрицательное значение.

Правила переопределения метода hashcode()

1. Если хеш-коды разные, то и входные объекты гарантированно разные.
2. Если хеш-коды равны, то входные объекты не всегда равны.
3. При вычислении хеш-кода следует использовать те же поля, которые сравниваются в equals и которые не вычисляются на основе других значений.

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

Следует 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 хеш-коды называется **коллизией**.

Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

Почему нельзя реализовать hashcode(), который будет гарантированно уникальным для каждого объекта?

В Java множество возможных хеш-кодов ограничено типом int, а множество объектов ничем не ограничено.

Из-за этого вполне возможна ситуация, что хеш-коды разных объектов могут совпасть.

*Почему хеш-код в виде 31 * x + y предпочтительнее чем x + y?*

- множитель создает зависимость значения хеш-кода от очередности обработки полей, что в итоге порождает лучшую хеш-функцию;
- 31 можно легко сдвигать побитово;
- в хеш-коде должны фигурировать поля, которые фигурируют в equals().

Чем a.getClass().equals(A.class) отличается от a instanceOf A.class?

getClass() получает только класс, а оператор **instanceof** проверяет, является ли объект экземпляром класса или его потомком.

instanceof

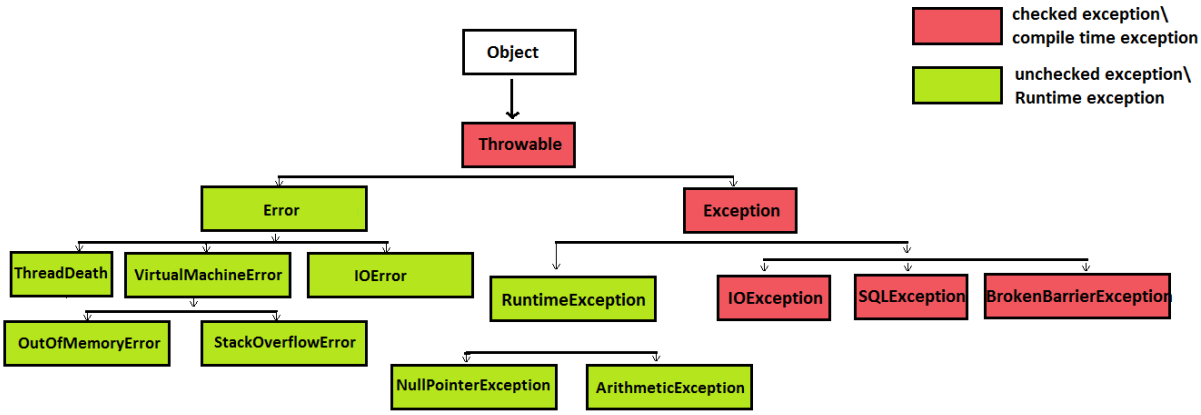
Оператор instanceof сравнивает объект и указанный тип. Его можно использовать для проверки, является ли данный объект экземпляром некоторого класса, либо экземпляром его дочернего класса, либо экземпляром класса, который реализует указанный интерфейс.

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

Что такое исключение?

Исключение – **это ошибка (является объектом),** возникающая во время выполнения программы.

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 серьезные 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.

Расскажите про обрабатываемые и необрабатываемые исключения

В Java все исключения делятся на два типа:

- **checked** (контролируемые/проверяемые исключения) должны обрабатываться блоком catch или описываться в сигнатуре метода (например throws IOException), наличие такого обработчика/модификатора сигнатуры проверяются 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 сигнатуре метода.

Можно ли обработать необрабатываемые исключения?

Можно, чтобы в некоторых случаях программа не прекратила работу. Отлавливаются в try-catch.

Какой оператор позволяет принудительно выбросить исключение?

Это оператор **throw**:

throw new Exception();

О чем говорит ключевое слово throws?

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

Передает обработку исключения вышестоящему методу. Используется в конструкторе, методе, классе.

Как написать собственное («пользовательское») исключение?

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

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 серьезные 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 ее 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 передается 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
}
```

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

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

Работает точно так же: после выхода из блока try выполняется блок finally.

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

Да, кроме случаев завершения работы программы или JVM:

- Finally может не выполниться в случае если в блоке try вызывает System.exit(0).
- Runtime.getRuntime().exit(0), Runtime.getRuntime().halt(0) и если во время исполнения блока try виртуальная машина выполнила недопустимую операцию и будет закрыта.
- В блоке try{} бесконечный цикл.

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

Может, исключение будет передано в виртуальную машину Java (JVM).

В каком порядке следует обрабатывать исключения в catch-блоках?

От наследника к предку.

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 создает секцию finally, в которой и происходит освобождение занятых в блоке ресурсов. Под ресурсами подразумеваются сущности, реализующие интерфейс **java.lang.Autocloseable**.

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

Что произойдет, если исключение будет выброшено из блока catch, после чего другое исключение будет выброшено из блока finally?

finally-секция может «перебить» throw/return при помощи другого throw/return.

Что произойдет, если исключение будет выброшено из блока catch, после чего другое исключение будет выброшено из метода close() при использовании try-with-resources?

В **try-with-resources** добавлена возможность хранения «подавленных» исключений, и брошенное try-блоком исключение **имеет больший приоритет**, **чем исключения**, получившиеся **во время закрытия.**

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 «сериализация» и как она реализована в Java?

Сериализация (Serialization) – процесс преобразования структуры данных в линейную последовательность байтов для дальнейшей передачи или сохранения. Сериализованные объекты можно затем восстановить (десериализовать).

В Java, согласно спецификации Java Object Serialization, существует два стандартных способа сериализации: стандартная сериализация через использование интерфейса **java.io.Serializable** и «расширенная» сериализация – **java.io.Externalizable.**

Сериализация позволяет в определенных пределах изменять класс. Вот наиболее важные изменения, с которыми спецификация Java Object Serialization может справляться автоматически:

- добавление в класс новых полей;
- изменение полей из статических в нестатические;
- изменение полей из транзитных в нетранзитные.

Обратные изменения (из нестатических полей в статические и из нетранзитных в транзитные) или удаление полей требуют определенной дополнительной обработки в зависимости от того, какая степень обратной совместимости необходима.

Для чего нужна сериализация?

Для компактного сохранения состояния объекта и считывание этого состояния.

Опишите процесс сериализации/десериализации с использованием Serializable

При использовании Serializable применяется алгоритм сериализации, который с помощью рефлексии (Reflection API) выполняет:

- запись в поток метаданных о классе, ассоциированном с объектом (имя класса, идентификатор SerialVersionUID, идентификаторы полей класса);
- рекурсивную запись в поток описания суперклассов до класса java.lang.Object (не включительно);
- запись примитивных значений полей сериализуемого экземпляра, начиная с полей самого верхнего суперкласса;
- рекурсивную запись объектов, которые являются полями сериализуемого объекта.

При этом ранее сериализованные объекты повторно не сериализуются, что позволяет алгоритму корректно работать с циклическими ссылками.

Для выполнения десериализации под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается. Однако при десериализации будет вызван конструктор без параметров родительского несериализуемого класса, а его отсутствие повлечет ошибку десериализации.

Как изменить стандартное поведение сериализации/десериализации?

Реализовать интерфейс **java.io.Externalizable**, который позволяет применение пользовательской логики сериализации. Способ сериализации и десериализации описывается в методах writeExternal() и readExternal(). Во время десериализации вызывается конструктор без параметров, а потом уже на созданном объекте вызывается метод readExternal.

Если у сериализуемого объекта реализован один из следующих методов, то механизм сериализации будет использовать его, а не метод по умолчанию :

- writeObject() – запись объекта в поток;
- readObject() – чтение объекта из потока;
- writeReplace() – позволяет заменить себя экземпляром другого класса перед записью;
- readResolve() – позволяет заменить на себя другой объект после чтения.

Какие поля не будут сериализованы при сериализации? Будет ли сериализовано final-поле?

Поля с модификатором **transient**. В таком случае после восстановления его значение будет null.

Поля **static**. Значения статических полей автоматически не сохраняются.

Поля с модификатором **final** сериализуются **как** и **обычные**. **За** одним **исключением** – их **невозможно десериализовать при** использовании **Externalizable**, поскольку final-поля должны быть инициализированы в конструкторе, а после этого в readExternal изменить значение этого поля будет невозможно. Соответственно, если необходимо сериализовать объект с final-полем неоходимо использовать только стандартную сериализацию **(Serializable за счет рефлексии)**.

Если final-поля не кастомные, то будут десериализовываться.

Как создать собственный протокол сериализации?

Для создания собственного протокола нужно **переопределить writeExternal**() и **readExternal**().

В отличие от двух других вариантов сериализации, здесь ничего не делается автоматически. Протокол полностью в наших руках.

Для создания собственного протокола сериализации достаточно реализовать интерфейс **Externalizable**, который содержит два метода:

public void writeExternal(ObjectOutput out) throws IOException;

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;

Какая роль поля serialVersionUID в сериализации?

serialVersionUID используется для указании версии сериализованных данных.

Если не объявить serialVersionUID в классе явно, среда выполнения Java делает это за нас, но этот процесс чувствителен ко многим метаданным класса, включая количество полей, тип полей, модификаторы доступа полей, интерфейсов, которые реализованы в классе и пр.

Рекомендуется явно объявлять serialVersionUID т. к. при добавлении, удалении атрибутов класса динамически сгенерированное значение может измениться и в момент выполнения будет выброшено исключение InvalidClassException.

private static final long serialVersionUID = 20161013L;

Когда стоит изменять значение поля serialVersionUID?

serialVersionUID нужно изменять при внесении в класс несовместимых изменений, например, при удалении какого-либо его атрибута.

В чем проблема сериализации Singleton?

Проблема в том, что после десериализации получим другой объект. Таким образом, сериализация дает возможность создать Singleton еще раз, что недопустимо.

Существует два способа избежать этого:

- явный запрет сериализации;
- определение метода с сигнатурой (default/public/private/protected/) Object readResolve() throws ObjectStreamException, назначением которого станет возврат замещающего объекта вместо объекта, на котором он вызван.

Как исключить поля из сериализации?

Для управления сериализацией при определении полей можно использовать ключевое слово **transient**, таким образом исключив поля из общего процесса сериализации.

Что обозначает ключевое слово transient?

Поля класса, помеченные модификатором transient, не сериализуются.

Обычно в таких полях хранится промежуточное состояние объекта, которое, к примеру, проще вычислить. Другой пример такого поля – ссылка на экземпляр объекта, который не требует сериализации или не может быть сериализован.

Какое влияние оказывают на сериализуемость модификаторы полей static и final?

При стандартной сериализации поля, имеющие модификатор static, не сериализуются. Соответственно, после десериализации это поле значения не меняет. При использовании реализации Externalizable сериализовать и десериализовать статическое поле можно, но не рекомендуется этого делать, т. к. это может сопровождаться трудноуловимыми ошибками.

Поля с модификатором final сериализуются как и обычные. За одним исключением – их невозможно десериализовать при использовании Externalizable, поскольку final-поля должны быть инициализированы в конструкторе, а после этого в readExternal() изменить значение этого поля будет невозможно. Соответственно, если необходимо сериализовать объект с final-полем, необходимо использовать только стандартную сериализацию.

Как не допустить сериализацию?

Чтобы не допустить автоматическую сериализацию, можно переопределить private методы для создания исключительной ситуации NotSerializableException.

```
private void writeObject(ObjectOutputStream out) throws IOException {
   throw new NotSerializableException();
}

private void readObject(ObjectInputStream in) throws IOException {
   throw new NotSerializableException();
}
```

Любая попытка записать или прочитать этот объект теперь приведет к возникновению исключительной ситуации.

Какие существуют способы контроля за значениями десериализованного объекта?

Если есть необходимость выполнения контроля за значениями десериализованного объекта, то можно использовать интерфейс ObjectInputValidation с переопределением метода validateObject().

Если вызвать метод validateObject() после десериализации объекта, то будет вызвано исключение InvalidObjectException при значении возраста за пределами 39...60.

```
public class Person implements java.io.Serializable,
```

```
                    java.io.ObjectInputValidation {
    ...
    @Override
    public void validateObject() throws InvalidObjectException {
        if ((age < 39) || (age > 60))
            throw new InvalidObjectException("Invalid age");
    }
}
```

Также существуют способы подписывания и шифрования, позволяющие убедиться, что данные не были изменены:

- с помощью описания логики в writeObject() и readObject();
- поместить в оберточный класс javax.crypto.SealedObject и/или java.security.SignedObject. Данные классы являются сериализуемыми, поэтому при оборачивании объекта в SealedObject создается подобие «подарочной упаковки» вокруг исходного объекта. Для шифрования необходимо создать симметричный ключ, управление которым должно осуществляться отдельно. Аналогично для проверки данных можно использовать класс SignedObject, для работы с которым также нужен симметричный ключ, управляемый отдельно.

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 оберток 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 оригинальное поле и его 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(). 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, который 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 переопределенного метода clone() у не Cloneable объекта вызовет выбрасывание **CloneNotSupportedException**.

Как создать глубокую копию объекта (2 способа)?

Глубокое клонирование требует выполнения следующих правил:

- нет необходимости копировать отдельно примитивные данные;
- все классы-члены в оригинальном классе должны поддерживать клонирование, Для каждого члена класса должен вызываться super.clone() при переопределении метода 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.

Сериализация – это еще один способ глубокого копирования. Просто сериализуем нужный объект и десериализуем его. При этом объект должен поддерживать интерфейс **Serializable**. Сохраняем объект в массив байт и потом читаем из него.

Рефлексия

Рефлексия (Reflection) – это механизм получения данных о программе во время ее 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.

Класс Optional

Опциональное значение Optional – это контейнер для объекта, который может содержать или не содержать значение null. Такая обертка является удобным средством предотвращения **NullPointerException**, т. к. имеет некоторые функции высшего порядка, избавляющие от добавления повторяющихся проверок if null/notNull.

Core-2

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 raw type (сырой тип)?

Это имя интерфейса без указания параметризованного типа:

List list = new ArrayList(); // raw type

List<Integer> listIntgrs = new ArrayList<>(); // parameterized type

Что такое стирание типов?

Суть заключается в том, что внутри класса не хранится никакой информации о типе-параметре. Эта информация доступна только на этапе компиляции и стирается (становится недоступной) в runtime.

В чем заключается разница между IO и NIO?

Java IO (input-output) является потокоориентированным, а Java NIO (new/non-blocking io) – буфер-ориентированным. Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кешируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. В Java NIO данные сначала считываются в буфер, что дает больше гибкости при обработке данных.

Потоки ввода/вывода в Java IO являются блокирующими. Это значит, что когда в потоке выполнения вызывается read() или write() метод любого класса из пакета java.io.*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого. Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным, пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим. То же самое справедливо и для неблокирующего вывода. Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом, пока они не будут полностью записаны.

В Java NIO имеются селекторы, которые позволяют одному потоку выполнения мониторить несколько каналов ввода. Т. е. существует возможность зарегистрировать несколько каналов с селектором, а потом использовать один поток выполнения для обслуживания каналов, имеющих доступные для обработки данные, или для выбора каналов, готовых для записи.

Какие классы поддерживают чтение и запись потоков в сжатом формате?

- DeflaterOutputStream – компрессия данных в формате deflate;
- Deflater – компрессия данных в формат ZLIB;
- ZipOutputStream – потомок DeflaterOutputStream для компрессии данных в формате Zip;
- GZIPOutputStream – потомок DeflaterOutputStream для компрессии данных в формат GZIP;
- InflaterInputStream – декомпрессия данных в формате deflate;
- Inflater – декомпрессия данных в формате ZLIB;
- ZipInputStream – потомок InflaterInputStream для декомпрессии данных в формате Zip;
- GZIPInputStream – потомок InflaterInputStream для декомпрессии данных в формате GZIP.

Что такое «каналы»?

Каналы (channels) – это логические (не физические) порталы, абстракции объектов более низкого уровня файловой системы (например, отображенные в памяти файлы и блокировки файлов), через которые осуществляется ввод/вывод данных, а буферы являются источниками или приемниками этих переданных данных. При организации вывода данные, которые необходимо отправить, помещаются в буфер, который затем передается в канал. При вводе данные из канала помещаются в заранее предоставленный буфер.

Каналы напоминают трубопроводы, по которым эффективно транспортируются данные между буферами байтов и сущностями по ту сторону каналов. Каналы – это шлюзы, которые позволяют получить доступ к сервисам ввода/вывода операционной системы с минимальными накладными расходами, а буферы – внутренние конечные точки этих шлюзов, используемые для передачи и приема данных.

Назовите основные классы потоков ввода/вывода?

Разделяют два вида потоков ввода/вывода:

- байтовые – java.io.InputStream, java.io.OutputStream;
- символьные – java.io.Reader, java.io.Writer.

В каких пакетах расположены классы потоков ввода/вывода?

java.io, java.nio. Для работы с потоками сжатых данных используются классы из пакета java.util.zip.

Какие подклассы класса InputStream вы знаете, для чего они предназначены?

- InputStream – абстрактный класс, описывающий поток ввода;
- BufferedInputStream – буферизованный входной поток;
- ByteArrayInputStream позволяет использовать буфер в памяти (массив байтов) в качестве источника данных для входного потока;
- DataInputStream – входной поток для байтовых данных, включающий методы для чтения стандартных типов данных Java;

- FileInputStream – входной поток для чтения информации из файла;
- FilterInputStream – абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства;
- ObjectInputStream – входной поток для объектов;
- StringBufferInputStream превращает строку (String) во входной поток данных InputStream;
- PipedInputStream реализует понятие входного канала;
- PrintStream – выходной поток, включающий методы print() и println();
- PushbackInputStream – разновидность буферизации, обеспечивающая чтение байта с последующим его возвратом в поток, позволяет «заглянуть» во входной поток и увидеть, что оттуда поступит в следующий момент, не извлекая информации;
- SequenceInputStream используется для слияния двух или более потоков InputStream в единый.

Для чего используется PushbackInputStream?

Разновидность буферизации, обеспечивающая чтение байта с последующим его возвратом в поток. Класс PushbackInputStream позволяет «заглянуть» во входной поток и увидеть, что оттуда поступит в следующий момент, не извлекая информации.

У класса есть дополнительный метод unread().

Для чего используется SequenceInputStream?

Класс SequenceInputStream позволяет сливать вместе несколько экземпляров класса InputStream. Конструктор принимает в качестве аргумента либо пару объектов класса InputStream, либо интерфейс Enumeration.

Во время работы класс выполняет запросы на чтение из первого объекта класса InputStream и до конца, а затем переключается на второй. При использовании интерфейса работа продолжится по всем объектам класса InputStream. По достижении конца связанный с ним поток закрывается. Закрытие потока, созданного объектом класса SequenceInputStream, приводит к закрытию всех открытых потоков.

Какой класс позволяет читать данные из входного байтового потока в формате примитивных типов данных?

Класс DataInputStream представляет поток ввода и предназначен для записи данных примитивных типов, таких, как int, double и т. д. Для каждого примитивного типа определен свой метод для считывания:

- boolean readBoolean(): считывает из потока булевое однобайтовое значение;
- byte readByte(): считывает из потока 1 байт;
- char readChar(): считывает из потока значение char;
- double readDouble(): считывает из потока 8-байтовое значение double;
- float readFloat(): считывает из потока 4-байтовое значение float;
- int readInt(): считывает из потока целочисленное значение int;

- long readLong(): считывает из потока значение long;
- short readShort(): считывает значение short;
- String readUTF(): считывает из потока строку в кодировке UTF-8.

Какие подклассы класса OutputStream вы знаете, для чего они предназначены?

- OutputStream – это абстрактный класс, определяющий потоковый байтовый вывод;
- BufferedOutputStream – буферизированный выходной поток;
- ByteArrayOutputStream – все данные, посылаемые в этот поток, размещаются в предварительно созданном буфере;
- DataOutputStream – выходной поток байт, включающий методы для записи стандартных типов данных Java;
- FileOutputStream – запись данных в файл на физическом носителе;
- FilterOutputStream – абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства;
- ObjectOutputStream – выходной поток для записи объектов;
- PipedOutputStream реализует понятие выходного канала.

Какие подклассы класса Reader вы знаете, для чего они предназначены?

- Reader – абстрактный класс, описывающий символьный ввод;
- BufferedReader – буферизованный входной символьный поток;
- CharArrayReader – входной поток, который читает из символьного массива;
- FileReader – входной поток, читающий файл;
- FilterReader – абстрактный класс, предоставляющий интерфейс для классов-надстроек;
- InputStreamReader – входной поток, транслирующий байты в символы;
- LineNumberReader – входной поток, подсчитывающий строки;
- PipedReader – входной канал;
- PushbackReader – входной поток, позволяющий возвращать символы обратно в поток;
- StringReader – входной поток, читающий из строки.

Какие подклассы класса Writer вы знаете, для чего они предназначены?

- Writer – абстрактный класс, описывающий символьный вывод;
- BufferedWriter – буферизованный выходной символьный поток;
- CharArrayWriter – выходной поток, который пишет в символьный массив;
- FileWriter – выходной поток, пишущий в файл;
- FilterWriter – абстрактный класс, предоставляющий интерфейс для классов-надстроек;

- OutputStreamWriter – выходной поток, транслирующий байты в символы;
- PipedWriter – выходной канал;
- PrintWriter – выходной поток символов, включающий методы print() и println();
- StringWriter – выходной поток, пишущий в строку;

В чем отличие класса PrintWriter от PrintStream?

В классе PrintWriter применен усовершенствованный способ работы с символами Unicode и другой механизм буферизации вывода. В классе PrintStream буфер вывода сбрасывался всякий раз, когда вызывался метод print() или println(), а при использовании класса PrintWriter существует возможность отказаться от автоматического сброса буферов, выполняя его явным образом при помощи метода flush().

Кроме того, методы класса PrintWriter никогда не создают исключений. Для проверки ошибок необходимо явно вызвать метод checkError().

Чем отличаются и что общего у InputStream, OutputStream, Reader, Writer?

- InputStream и его наследники – совокупность для получения байтовых данных из различных источников;
- OutputStream и его наследники – набор классов определяющих потоковый байтовый вывод;
- Reader и его наследники определяют потоковый ввод символов Unicode;
- Writer и его наследники определяют потоковый вывод символов Unicode.

Какие классы позволяют преобразовать байтовые потоки в символьные и обратно?

- OutputStreamWriter – «мост» между классом OutputStream и классом Writer, символы, записанные в поток, преобразовываются в байты;
- InputStreamReader – аналог для чтения, при помощи методов класса Reader читаются байты из потока InputStream и далее преобразуются в символы.

Какие классы позволяют ускорить чтение/запись за счет использования буфера?

- BufferedInputStream(InputStream in)/BufferedInputStream(InputStream in, int size);
- BufferedOutputStream(OutputStream out)/BufferedOutputStream(OutputStream out, int size);
- BufferedReader(Reader r)/BufferedReader(Reader in, int sz);
- BufferedWriter(Writer out)/BufferedWriter(Writer out, int sz).

Существует ли возможность перенаправить потоки стандартного ввода/вывода?

Класс System позволяет вам перенаправлять стандартный ввод, вывод и поток вывода ошибок, используя простой вызов статического метода:

- setIn(InputStream) – для ввода;

- setOut(PrintStream) – для вывода;
- setErr(PrintStream) – для вывода ошибок.

Какой класс предназначен для работы с элементами файловой системы?

File работает непосредственно с файлами и каталогами. Данный класс позволяет создавать новые элементы и получать информацию существующих: размер, права доступа, время и дату создания, путь к родительскому каталогу.

Какие методы класса File вы знаете?

Наиболее используемые методы класса File:

- boolean createNewFile(): делает попытку создать новый файл;
- boolean delete(): делает попытку удалить каталог или файл;
- boolean mkdir(): делает попытку создать новый каталог;
- boolean renameTo(File dest): делает попытку переименовать файл или каталог;
- boolean exists(): проверяет, существует ли файл или каталог;
- String getAbsolutePath(): возвращает абсолютный путь для пути, переданного в конструктор объекта;
- String getName(): возвращает краткое имя файла или каталога;
- String getParent(): возвращает имя родительского каталога;
- boolean isDirectory(): возвращает значение true, если по указанному пути располагается каталог;
- boolean isFile(): возвращает значение true, если по указанному пути находится файл;
- boolean isHidden(): возвращает значение true, если каталог или файл являются скрытыми;
- long length(): возвращает размер файла в байтах;
- long lastModified(): возвращает время последнего изменения файла или каталога;
- String[] list(): возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге;
- File[] listFiles(): возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге.

Что вы знаете об интерфейсе FileFilter?

Интерфейс FileFilter применяется для проверки, попадает ли объект File под некоторое условие. Этот интерфейс содержит единственный метод boolean accept(File pathName). Этот метод необходимо переопределить и реализовать. Например:

```
public boolean accept(final File file) {
   return file.isExists() && file.isDirectory();
}
```

Как выбрать все элементы определенного каталога по критерию (например, с определенным расширением)?

Метод File.listFiles() возвращает массив объектов File, содержащихся в каталоге. Метод может принимать в качестве параметра объект класса, реализующего FileFilter. Это позволяет включить в список только те элементы, для которых метод accept возвращает true (критерием может быть длина имени файла или его расширение).

Что вы знаете о RandomAccessFile?

Класс java.io.RandomAccessFile обеспечивает чтение и запись данных в произвольном месте файла. Он не является частью иерархии InputStream или OutputStream. Это полностью отдельный класс, имеющий собственные (в большинстве своем native) методы. Объяснением этого может быть то, что RandomAccessFile имеет во многом отличающееся поведение по сравнению с остальными классами ввода/вывода, так как позволяет в пределах файла перемещаться вперед и назад.

RandomAccessFile имеет такие специфические методы как:

- getFilePointer() для определения текущего местоположения в файле;
- seek() для перемещения на новую позицию в файле;
- length() для выяснения размера файла;
- setLength() для установки размера файла;
- skipBytes() для того, чтобы попытаться пропустить определенное число байт;
- getChannel() для работы с уникальным файловым каналом, ассоциированным с заданным файлом;
- методы для выполнения обычного и форматированного вывода из файла (read(), readInt(), readLine(), readUTF() и т.п .);
- методы для обычной или форматированной записи в файл с прямым доступом (write(), writeBoolean(), writeByte() и т. п.).

Следует отметить, что конструкторы RandomAccessFile требуют второй аргумент, указывающий необходимый режим доступа к файлу – только чтение ("r"), чтение и запись ("rw") или иную их разновидность.

Какие режимы доступа к файлу есть у RandomAccessFile?

«r» открывает файл только для чтения. Запуск любых методов записи данных приведет к выбросу исключения IOException.

«rw» открывает файл для чтения и записи. Если файл еще не создан, то осуществляется попытка создать его.

«rws» открывает файл для чтения и записи подобно «rw», но требует от системы при каждом изменении содержимого файла или метаданных синхронно записывать эти изменения на физический носитель.

«rwd» открывает файл для чтения и записи подобно «rws», но требует от системы синхронно записывать изменения на физический носитель только при каждом изменении содержимого файла. Если изменяются метаданные, синхронная запись не требуется.

Какой символ является разделителем при указании пути в файловой системе?

Для различных операционных систем символ разделителя различается. Для Windows это \, для Linux – /.

В Java получить разделитель для текущей операционной системы можно через обращение к статическому полю File.separator.

Что такое «абсолютный путь» и «относительный путь»?

Абсолютный (полный) путь – это путь, который указывает на одно и то же место в файловой системе вне зависимости от текущей рабочей директории или других обстоятельств. Полный путь всегда начинается с корневого каталога.

Относительный путь представляет собой путь по отношению к текущему рабочему каталогу пользователя или активного приложения.

Что такое «символьная ссылка»?

Символьная (символическая) ссылка (также «симлинк», Symbolic link) – специальный файл в файловой системе, в котором вместо пользовательских данных содержится путь к файлу, который должен быть открыт при попытке обратиться к данной ссылке (файлу). Целью ссылки может быть любой объект: например, другая ссылка, файл, каталог или даже несуществующий файл (в последнем случае при попытке открыть его должно выдаваться сообщение об отсутствии файла).

Символьные ссылки используются для более удобной организации структуры файлов на компьютере, так как:

- позволяют для одного файла или каталога иметь несколько имен и различных атрибутов;
- свободны от некоторых ограничений, присущих жестким ссылкам (последние действуют только в пределах одной файловой системы (одного раздела) и не могут ссылаться на каталоги).

Что такое default-методы интерфейса?

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:

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:

Paper.show();

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 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.

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.

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 с доступом 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 конструктор.

Объясните выражение System.out::println.

Данное выражение иллюстрирует механизм instance method reference: передачи ссылки на метод println() статического поля out класса System.

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, можно строить цепочки из нескольких 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.

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, 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 распространенных 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.

Также 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 используются несколько потоков процессора. Параллельные стримы используют общий ForkJoinPool, доступный через статический метод ForkJoinPool.commonPool(). 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 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. Также 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 сколько-нибудь долгих операций (например, сетевых соединений), так как все параллельные стримы работают с 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, 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() представляет результат 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 определенное 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() создает 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, создает 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 ключом 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 ее учетом.

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());

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.

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";
}
```

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

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».

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.

List vs. Set

Разница между списком и множеством в Java:

Список – это упорядоченная последовательность элементов, тогда как Set – это отдельный список элементов, который не упорядочен.

Список допускает дублирование, а Set не допускает дублирование элементов.

Список разрешает любое количество нулевых значений в своей коллекции, а Set разрешает только одно нулевое значение в своей коллекции.

Список может быть вставлен как в прямом, так и в обратном направлении с помощью Listiterator, тогда как Set можно просматривать только в прямом направлении с помощью итератора.

Map не в Collection

Интерфейс Map реализован классами:

- Hashtable – хеш-таблица, методы которой синхронизированы. Не позволяет использовать null в качестве значения или ключа и не является упорядоченной.
- HashMap – хеш-таблица. Позволяет использовать null в качестве значения или ключа и не является упорядоченной.
- LinkedHashMap – упорядоченная реализация хеш-таблицы.
- TreeMap – реализация, основанная на красно-черных деревьях. Является упорядоченной и предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator либо сохраняет элементы с использованием «natural ordering».
- WeakHashMap – реализация хеш-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 жестких ссылок).

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.

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);
- на добавление и удаление в начало или конец списка потребуется константное O(1);
- вставка или удаление в/из произвольного место линейное 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.

См. Чем отличается ArrayList от LinkedList.

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 в 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.

См. Чем отличается ArrayList от LinkedList.

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.

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 для хранения 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). Значения от -128 до 127 кешируются, 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 = 40 байт и 24 байта. Итого 64 байта.

Оцените количество памяти для хранения 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.

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, вызвав метод descendingIterator().

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 с помощью методов итератора.

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

- при изменении коллекции счетчик модификаций также 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.

Как поведет 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 поведет 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 другой итератор, работающий по принципу fail-safe, к примеру, для List можно использовать ListIterator;
- 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.

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.

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 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.

Comparator vs. Comparable

Интерфейс Comparable является хорошим выбором, когда он используется для определения порядка по умолчанию или, другими словами, если это основной способ сравнения объектов.

Зачем же использовать Comparator, если уже есть Comparable?

Есть несколько причин:

- иногда нельзя изменить исходный код класса, чьи объекты необходимо отсортировать, что делает невозможным использование Comparable;
- использование компараторов позволяет избежать добавления дополнительного кода в классы домена;
- можно определить несколько разных стратегий сравнения, что невозможно при использовании Comparable.

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 можно модифицировать список, добавляя/удаляя элементы с помощью методов add() и remove(), Iterator не поддерживает данного функционала.

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.

Сравните интерфейсы Queue и Deque. 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.

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 задается 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 определенного свойства.

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.

Как устроен HashMap?

HashMap состоит из «корзин» (buckets). С технической точки зрения «корзины» – это элементы массива, которые хранят ссылки на списки элементов. При добавлении новой пары «ключ-значение» вычисляется хеш-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 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 хеш-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 хеш-таблице не может превышать размера массива: по мере увеличения числа элементов и повышения коэффициента заполнения производительность структуры резко падает, поэтому необходимо проводить перехеширование;
- сложно организовать удаление элемента;
- первые два метода открытой адресации приводят к проблеме первичной и вторичной группировок.

Преимущества хеш-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 хеш-функция распределяет элементы по корзинам равномерно, временная сложность станет не хуже логарифмического времени O(log(N)), а в случае, когда хеш-функция постоянно возвращает одно и то же значение, HashMap превратится в связный список со сложностью O(n).

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 хеш-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 хеш-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?

Хеш-код массива не зависит от хранимых в нем элементов, а присваивается при создании массива (метод вычисления хеш-кода массива не переопределен и вычисляется по стандартному Object.hashCode() на основании адреса массива). Также 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 2^{32}.

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 определенным ключом, может потребоваться перебор всего списка.

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 хеш-кода ключа; определение номера корзины; поиск значения; возврат значения.

Сколько создается новых объектов, когда 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 хеш-таблице в момент ее создания.

LoadFactor – коэффициент заполнения HashMap, при превышении которого происходит увеличение количества корзин и автоматическое перехеширование. Равен отношению числа уже хранимых элементов в таблице к ее 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 разница между HashMap и IdentityHashMap? Для чего нужна IdentityHashMap?

IdentityHashMap – это структура данных, также 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 него 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 хеш-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.

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 объекта 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 хеш-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.

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 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 гораздо быстрей.

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

LinkedHashMap – что в нем от LinkedList, а что от HashMap?

Реализация LinkedHashMap отличается от HashMap поддержкой двухсвязного 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.

NavigableSet

Интерфейс унаследован от SortedSet и расширяет методы навигации, находя ближайшее совпадение по заданному значению. Как и в родительском интерфейсе, в NavigableSet не может быть дубликатов.

Многопоточка

Чем процесс отличается от потока?

Процесс – экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). **Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого**.

Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Поток (thread) – **способ выполнения процесса, определяющий последовательность исполнения кода в процессе**. **Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах**.

Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах. Так как потоки расходуют существенно меньше ресурсов, чем процессы, в ходе выполнения работы выгоднее создавать дополнительные потоки и избегать создания новых процессов.

Чем Thread отличается от Runnable? Когда нужно использовать Thread, а когда Runnable?

Thread – это класс, некоторая надстройка над физическим потоком.

Runnable – это интерфейс, представляющий абстракцию над выполняемой задачей.

Помимо того, что **Runnable помогает разрешить проблему множественного наследования**, несомненный плюс от его использования состоит в том, что он **позволяет отделить логику выполнения задачи от непосредственного управления потоком.**

В классе **Thread** имеется несколько методов, которые можно переопределить в порожденном классе. Из них обязательному переопределению подлежит только метод **run**(). Этот же метод должен быть определен и при реализации интерфейса **Runnable**. Некоторые программисты считают, что создавать подкласс, порожденный от класса **Thread**, следует только в том случае, если нужно дополнить его новыми функциями. Так, если переопределять любые другие методы из класса **Thread** не нужно, то можно ограничиться только реализацией интерфейса **Runnable**. Кроме того, реализация интерфейса **Runnable** позволяет создаваемому потоку наследовать класс, отличающийся от **Thread**.

Что такое монитор? Как монитор реализован в java?

Монитор – механизм синхронизации потоков, обеспечивающий доступ к неразделяемым ресурсам. Частью монитора является mutex, который встроен в класс Object и имеется у каждого объекта.

Удобно представлять mutex как id захватившего его объекта. Если id равен 0 – ресурс свободен. Если не 0 – ресурс занят. Можно встать в очередь и ждать его освобождения.

В Java **монитор реализован** с помощью ключевого слова **synchronized**.

Что такое синхронизация? Какие способы синхронизации существуют в Java?

Синхронизация – это **процесс**, **который позволяет выполнять потоки параллельно.**

В Java все объекты имеют блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта.

Способы синхронизации в Java:

1. Системная синхронизация с использованием wait()/notify().

Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод wait(), предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод notify(), предварительно захватив монитор объекта, в результате чего ждущий на объекте поток «просыпается» и продолжает свое выполнение. В обоих случаях монитор надо захватывать в явном виде через synchronized-блок, потому как методы wait()/notify() не синхронизированы!

2. Системная синхронизация с использованием join().

Метод join(), вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связанный с этим экземпляром, закончит работу.

3. Использование классов из пакета java.util.concurrent.Locks – механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: **Lock, Condition, ReadWriteLock.**

Как работают методы wait(), notify() и notifyAll()?

- **wait(): освобождает монитор** и **переводит вызывающий поток** в состояние **ожидания** до тех пор, **пока другой поток не вызовет метод notify()/notifyAll();**
- **notify(): продолжает** работу **потока**, у которого ранее был вызван метод wait();
- **notifyAll(): возобновляет** работу **всех потоков**, у которых ранее был вызван метод wait().

Когда вызван метод wait(), поток освобождает блокировку на объекте и переходит из состояния «работающий» (running) в состояние «ожидание» (waiting). Метод **notify() подает сигнал** одному из **потоков**, **ожидающих на объекте**, **чтобы перейти в состояние «работоспособный» (runnable)**. При этом невозможно определить, какой из ожидающих потоков должен стать работоспособным. Метод notifyAll() заставляет все ожидающие потоки для объекта вернуться в состояние «работоспособный» (runnable). **Если ни один поток не находится в ожидании на методе wait(), то при вызове notify() или notifyAll() ничего не происходит.**

wait(), notify() и notifyAll() **должны** вызываться **только из синхронизированного кода.**

В каких состояниях может находиться поток?

New – объект класса Thread создан, но еще не запущен. Он еще не является потоком выполнения и естественно не выполняется.

Runnable – поток готов к выполнению, но планировщик еще не выбрал его.

Running – поток выполняется.

Waiting/blocked/sleeping – поток блокирован или ждет окончания работы другого потока.

Dead – поток завершен. Будет выброшено исключение при попытке вызвать метод start() для потока dead.

Что такое семафор? Как он реализован в Java?

Semaphore – это новый тип синхронизатора: семафор со счетчиком, реализующий шаблон синхронизации «Семафор». Доступ управляется с помощью счетчика: изначальное значение счетчика задается в конструкторе при создании синхронизатора, когда поток заходит в заданный блок кода, то значение счетчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счетчика равно нулю, то текущий поток блокируется, пока кто-нибудь не выйдет из защищаемого блока. Semaphore используется для защиты дорогих ресурсов, которые доступны в ограниченном количестве, например, подключение к базе данных в пуле.

Что означает ключевое слово volatile? Почему операции над volatile переменными не атомарны?

Переменная volatile является атомарной для чтения, но операции над переменной НЕ являются атомарными. Поля, для которых неприемлемо увидеть «несвежее» (stale) значение в результате кеширования или переупорядочения.

Если происходит какая-то операция, например, инкримент, то атомарность уже не обеспечивается, потому что сначала выполняется чтение(1), потом изменение(2) в локальной памяти, а затем запись(3). Такая операция не является атомарной и в нее может вклиниться поток по середине.

Атомарная операция выглядит единой и неделимой командой процессора.

Переменная volatile находится в хипе, а не в кеше стека.

Для чего нужны типы данных atomic? Чем отличаются от volatile?

volatile не гарантирует атомарность. Например, операция count++ не станет атомарной просто потому, что count объявлена volatile. С другой стороны, **class AtomicInteger** предоставляет **атомарный метод для выполнения таких комплексных операций атомарно**, например getAndIncrement() – **атомарная замена оператора инкремента,** его можно использовать, чтобы атомарно увеличить текущее значение на один. Похожим образом сконструированы атомарные версии и для других типов данных.

Что-то наподобие обертки над примитивами.

Что такое потоки демоны? Для чего они нужны? Как создать поток-демон?

Потоки-демоны **работают в фоновом режиме вместе с программой**, но не являются неотъемлемой частью программы**.**

Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон с помощью метода **setDaemon(boolean value)**, вызванного у потока до его запуска.

Метод boolean isDaemon() позволяет определить, является ли указанный поток демоном или нет. **Основной поток приложения может завершить выполнение потока-демона** (в отличие от обычных потоков) **с окончанием кода метода main(), не обращая внимания**, **что поток-демон еще работает.**

Поток демон можно сделать, только если он еще не запущен.

Пример демона – сборщик мусора (GC).

Что такое приоритет потока? На что он влияет? Какой приоритет у потоков по умолчанию?

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда и какому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета.

Чтобы установить приоритет потока, используется метод класса Thread: final void setPriority(int level). Значение level изменяется в пределах от Thread.MIN_PRIORITY = 1 до Thread.MAX_PRIORITY = 10. Приоритет по умолчанию – Thread.NORM_PRIORITY = 5.

Получить текущее значение приоритета потока можно, вызвав метод: final int getPriority() у экземпляра класса Thread.

Метод yield() можно использовать для того, чтобы принудить планировщик выполнить другой поток, который ожидает своей очереди.

Как работает Thread.join()? Для чего он нужен?

Когда поток вызывает join(), он будет ждать, пока поток, к которому он присоединяется, будет завершен либо отработает переданное время:

void join()

void join(long millis) – с временем ожидания

void join(long millis, int nanos)

Применение: при распараллеливании вычисления, надо дождаться результатов, чтобы собрать их в кучу и продолжить выполнение.

Чем отличаются методы wait() и sleep()?

Метод sleep() приостанавливает поток на указанное время. Состояние меняется на WAITING, по истечении – RUNNABLE. **Монитор не освобождается**.

Метод wait() меняет состояние потока на WAITING. Может быть вызван только у объекта, владеющего блокировкой, в противном случае выкинется исключение **IllegalMonitorStateException.**

Можно ли вызвать start() для одного потока дважды?

Нельзя стартовать поток больше одного раза. Поток не может быть перезапущен, если он уже завершил выполнение.

Выдает: **IllegalThreadStateException**.

Как правильно остановить поток? Для чего нужны методы stop(), interrupt(), interrupted(), isInterrupted()?

Как остановить поток:

На данный момент в Java принят уведомительный порядок остановки потока (хотя JDK 1.0 и имеет несколько управляющих выполнением потока методов, например stop(), suspend() и resume() – в следующих версиях JDK все они были помечены как deprecated из-за потенциальных угроз взаимной блокировки).

1. Для корректной остановки потока можно использовать метод класса Thread **interrupt().** Этот метод выставляет внутренний флаг-статус прерывания. В дальнейшем состояние этого флага можно проверить с помощью метода isInterrupted() или Thread.interrupted() (для текущего потока). **Метод interrupt() способен вывести поток из состояния ожидания или спячки**. Т. е., если у потока были вызваны методы sleep() или wait(), текущее состояние прервется и будет выброшено исключение InterruptedException. Флаг в этом случае не выставляется.

Схема действия при этом получается следующей:

- реализовать поток;
- в потоке периодически проводить проверку статуса прерывания через вызов isInterrupted();
- если состояние флага изменилось или было выброшено исключение во время ожидания/спячки, следовательно поток пытаются остановить извне;
- принять решение – продолжить работу (если по каким-то причинам остановиться невозможно) или освободить заблокированные потоком ресурсы и закончить выполнение.

Возможная проблема, которая присутствует в этом подходе – блокировки на потоковом вводе-выводе. Если поток заблокирован на чтении данных – вызов interrupt() из этого состояния его не выведет. Решения тут различаются в зависимости от типа источника данных. Если чтение идет из файла, то долговременная блокировка крайне маловероятна и тогда можно просто дождаться выхода из метода read(). Если же чтение каким-то образом связано с сетью, то стоит использовать неблокирующий ввод-вывод из Java NIO.

2. Второй вариант реализации метода остановки (а также и приостановки) – сделать собственный аналог interrupt(). Т. е. объявить в классе потока флаги на остановку и/или приостановку и выставлять их путем вызова заранее определенных методов извне. Методика действия при этом остается прежней – проверять установку флагов и принимать решения при их изменении.

Недостатки такого подхода:

- потоки в состоянии ожидания таким способом не «оживить»;
- выставление флага одним потоком совсем не означает, что второй поток тут же его увидит, для увеличения производительности виртуальная машина использует кеш

данных потока, в результате чего обновление переменной у второго потока может произойти через неопределенный промежуток времени (хотя допустимым решением будет объявить переменную-флаг как volatile).

Почему не рекомендуется использовать метод Thread.stop()?

При принудительной остановке (приостановке) потока stop() прерывает поток в недетерменированном месте выполнения, в результате становится совершенно непонятно, что делать с принадлежащими ему ресурсами. Поток может открыть сетевое соединение – что в таком случае делать с данными, которые еще не вычитаны? Где гарантия, что после дальнейшего запуска потока (в случае приостановки) он сможет их дочитать? Если поток блокировал разделяемый ресурс, то как снять эту блокировку и не переведет ли принудительное снятие к нарушению консистентности системы? То же самое можно расширить и на случай соединения с базой данных: если поток остановят посередине транзакции, то кто ее будет закрывать? Кто и как будет разблокировать ресурсы?

В чем разница между interrupted() и isInterrupted()?

Механизм прерывания работы потока в Java реализован с использованием внутреннего флага, известного как статус прерывания. Прерывание потока вызовом Thread.interrupt() устанавливает этот флаг. Методы Thread.interrupted() и isInterrupted() позволяют проверить, является ли поток прерванным.

Когда прерванный поток проверяет статус прерывания, вызывая **статический метод Thread.interrupted(), статус прерывания сбрасывается.**

Нестатический метод isInterrupted() используется одним потоком для проверки статуса прерывания у другого потока, **не изменяя флаг прерывания.**

Чем Runnable отличается от Callable?

- интерфейс **Runnable** появился в Java 1.0, а интерфейс **Callable** был введен в Java 5.0 в составе библиотеки java.util.concurrent;
- **классы, реализующие** интерфейс **Runnable** для выполнения задачи **должны реализовывать метод run(), классы, реализующие интерфейс Callable – метод call();**
- метод Runnable.**run() не возвращает** никакого значения;
- Callable – это параметризованный функциональный интерфейс, Callable.**call() возвращает Object, если он не параметризован, иначе указанный тип;**
- метод **run() НЕ может выбрасывать проверяемые исключения, в то время как метод call() может.**

Что такое FutureTask?

FutureTask представляет собой **отменяемое асинхронное вычисление** в **параллельном потоке**. Этот **класс** предоставляет **базовую реализацию Future** с методами для запуска и остановки вычисления, методами для запроса состояния вычисления и извлечения результатов.

Результат может быть получен, только когда вычисление завершено, **метод получения будет заблокирован, если вычисление еще не завершено**.

Объекты FutureTask могут быть использованы для обертки объектов Callable и Runnable. Так как **FutureTask помимо Future реализует Runnable**, его можно передать в Executor на выполнение.

Что такое deadlock?

Взаимная блокировка (deadlock) – **явление,** при котором **все потоки** находятся **в режиме ожидания и свое состояние не меняют.** Происходит, когда достигаются состояния:

- **взаимного исключения:** по крайней мере **один** ресурс занят в режиме неделимости, и следовательно только один поток может использовать ресурс в данный момент времени;
- **удержания и ожидания:** поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсы, которые удерживаются другими потоками;
- **отсутствия предочистки:** операционная система не переназначает ресурсы – если они уже заняты, то должны отдаваться удерживающим потокам сразу же;
- **цикличного ожидания:** поток ждет освобождения ресурса другим потоком, который в свою очередь ждет освобождения ресурса заблокированного первым потоком.

Простейший способ **избежать взаимной блокировки** – **не допускать цикличного ожидания.** Этого **можно достичь**, **получая мониторы разделяемых ресурсов в определенном порядке и освобождая их в обратном порядке.**

Что такое livelock?

livelock – **тип взаимной блокировки**, при котором **несколько потоков выполняют бесполезную работу**, **попадая** в **зацикленность при попытке получения** каких-либо **ресурсов**. При этом **их состояния** постоянно **изменяются** в **зависимости друг от друга.** Фактической **ошибки не возникает**, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock.

Узнать о наличии livelock можно, например, проверив уровень загрузки процессора в состоянии покоя.

Примеры livelock:

- 2 человека встречаются в узком коридоре и каждый, пытаясь быть вежливым, отходит в сторону, и так они бесконечно двигаются из стороны в сторону, абсолютно не продвигаясь в нужном им направлении;
- аналогично 2 пылесоса в узком коридоре пытаются выяснить, кто должен первым убрать один и тот же участок;
- на равнозначном перекрестке 4 автомобиля не могут определить, кто из них должен уступить дорогу;
- одновременный звонок друг другу.

Что такое race condition?

Состояние гонки (race condition) – **ошибка проектирования многопоточной системы или приложения**, **при которой работа зависит** от того**, в каком порядке выполняются потоки.** Состояние гонки **возникает, когда поток, который должен исполниться в начале, проиграл гонку, и первым исполняется другой поток:** поведение кода изменяется, из-за чего возникают недетерменированные ошибки.

DataRace – это свойство выполнения программы. Согласно JMM, выполнение считается содержащим гонку данных, если оно содержит по крайней мере два конфликтующих доступа (чтение или запись в одну и ту же переменную), которые не упорядочены отношениями «happens before».

Starvation – потоки не заблокированы, но есть нехватка ресурсов, из-за чего потоки ничего не делают.

Самый простой способ решения – копирование переменной в локальную переменную. Или просто синхронизация потоков методами и synchronized-блоками.

Что такое Фреймворк fork/join? Для чего он нужен?

Фреймворк Fork/Join, представленный в JDK 7, – это набор классов и интерфейсов, позволяющих использовать преимущества многопроцессорной архитектуры современных компьютеров. **Он разработан для выполнения задач, которые можно рекурсивно разбить на маленькие подзадачи, которые можно решать параллельно.**

Этап Fork: **большая задача разделяется** на **несколько меньших подзадач,** которые в свою очередь также разбиваются на меньшие. **И так до тех пор, пока задача не становится тривиальной и решаемой последовательным способом.**

Этап Join: далее (опционально) идет процесс «свертки» – **решения подзадач некоторым образом объединяются, пока не получится решение всей задачи.**

Решение всех подзадач (в т. ч. и само разбиение на подзадачи) происходит параллельно.

Для **решения некоторых** задач **этап Join не требуется. Например**, для параллельного QuickSort – массив рекурсивно делится на все меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из 1 элемента. Хотя в некотором смысле Join будет необходим и тут, т. к. все равно остается необходимость дождаться, пока не закончится выполнение всех подзадач.

Еще одно **преимущество** этого фреймворка заключается в том, что он **использует work-stealing алгоритм**: **потоки**, которые **завершили** выполнение **собственных подзадач**, могут «**украсть**» **подзадачи у** других **потоков**, **которые все еще заняты.**

Что означает ключевое слово synchronized? Где и для чего может использоваться?

Зарезервированное слово позволяет добиваться синхронизации в помеченных им **методах** или **блоках кода**.

Что является монитором у статического synchronized-метода?

Объект типа Class, соответствующий классу, в котором определен метод.

Что является монитором у нестатического synchronized-метода?

Объект this.

util.Concurrent поверхностно

Классы и интерфейсы пакета java.util.concurrent объединены в несколько групп по функциональному признаку:

collections – набор эффективно работающих в многопоточной среде коллекций CopyOnWriteArrayList(Set), ConcurrentHashMap.

Итераторы классов данного пакета представляют данные на определенный момент времени. Все операции по изменению коллекции (add, set, remove) приводят к созданию новой копии внутреннего массива. Этим гарантируется, что при проходе итератором по коллекции не будет ConcurrentModificationException.

Отличие **ConcurrentHashMap** связано с внутренней структурой хранения пар key-value. ConcurrentHashMap использует несколько сегментов, и данный класс нужно рассматривать как группу HashMap'ов. Количество сегментов по умолчанию равно 16. Если пара key-value хранится в 10-ом сегменте, то ConcurrentHashMap заблокирует при необходимости только 10-й сегмент, и не будет блокировать остальные 15.

CopyOnWriteArrayList:

- volatile массив снутри;
- lock только при модификации списка, поэтому операции чтения очень быстрые;
- новая копия массива при модификации;
- fail-fast итератор;
- модификация через iterator невозможна – UnsupportedOperationException.

synchronizers – объекты синхронизации, позволяющие разработчику управлять и/или ограничивать работу нескольких потоков. Содержит пять объектов синхронизации: semaphore, countDownLatch, ciclycBarrier, exchanger, phaser.

CountDownLatch – объект синхронизации потоков, блокирующий один или несколько потоков до тех пор, пока не будут выполнены определенные условия. Количество условий задается счетчиком. При обнулении счетчика, т. е. при выполнении всех условий, блокировки выполняемых потоков будут сняты и они продолжат выполнение кода. Одноразовый.

CyclicBarrier – барьерная синхронизация останавливает поток в определенном месте в ожидании прихода остальных потоков группы. Как только все потоки достигнут барьера, барьер снимается и выполнение потоков продолжается. Как и CountDownLatch, использует счетчик и похож на него. Отличие связано с тем, барьер можно использовать повторно (в цикле).

Exchanger – объект синхронизации, используемый для двустороннего обмена данными между двумя потоками. При обмене данными допускается null-значения, что позволяет использовать класс для односторонней передачи объекта или же просто, как синхронизатор двух потоков. Обмен данными выполняется вызовом метода exchange, сопровождаемый самоблокировкой потока. Как только второй поток вызовет метод exchange, то синхронизатор Exchanger выполнит обмен данными между потоками.

Phaser – объект синхронизации типа «Барьер», но в отличие от CyclicBarrier может иметь несколько барьеров (фаз), и количество участников на каждой фазе может быть разным.

atomic – набор атомарных классов для выполнения атомарных операций. Операция является атомарной, если ее можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized.

Queues содержит классы формирования неблокирующих и блокирующих очередей для многопоточных приложений. Неблокирующие очереди «заточены» на скорость выполнения, блокирующие очереди приостанавливают потоки при работе с очередью.

Locks – механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: Lock, Condition, ReadWriteLock.

Executors включает средства, называемые сервисами исполнения, позволяющие управлять потоковыми задачами с возможностью получения результатов через интерфейсы Future и Callable.

ExecutorService служит альтернативой классу Thread, предназначенному для управления потоками. В основу сервиса исполнения положен интерфейс Executor, в котором определен один метод:

void execute(Runnable thread);

При вызове метода execute исполняется поток thread.

Stream API & ForkJoinPool, как связаны, что это такое?

В Stream API есть простой способ распараллеливания потока методом **parallel()** или **parallelStream(**), чтобы получить выигрыш в производительности на многоядерных машинах.

По умолчанию parallel stream используют ForkJoinPool.commonPool. Этот пул создается статически и живет пока не будет вызван System::exit. Если задачам не указывать конкретный пул, то они будут исполняться в рамках commonPool.

По умолчанию размер пула равен на 1 меньше, чем количество доступных ядер.

Когда некий тред отправляет задачу в common pool, то пул может использовать вызывающий тред (caller-thread) в качестве воркера. ForkJoinPool пытается загрузить своими задачами и вызывающий тред.

Java Memory Model

Описывает как потоки должны взаимодействовать через общую память. Определяет набор действий межпоточного взаимодействия. В частности, чтение и запись переменной, захват и освобождение монитора, чтение и запись volatile переменной, запуск нового потока.

JMM определяет отношение между этими действиями «happens-before» – абстракцией, обозначающей, что если операция X связана отношением happens-before с операцией Y, то весь код следуемый за операцией Y, выполняемый в одном потоке, видит все изменения, сделанные другим потоком до операции X.

Можно выделить несколько основных областей, имеющих отношение к модели памяти:

Видимость (visibility). Один поток может временно сохранить значения некоторых полей не в основную память, а в регистры или локальный кеш процессора, таким образом второй поток, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кешами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

К вопросу видимости имеют отношение следующие ключевые слов языка Java: synchronized, volatile, final.

С точки зрения Java все переменные (за исключением локальных переменных, объявленных внутри метода) хранятся в heap-памяти, которая доступна всем потокам. Кроме этого каждый поток имеет локальную рабочую память, где он хранит копии переменных, с которыми он работает, и при выполнении программы поток работает только с этими копиями.

Переупорядочивание (reordering). Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Процессор может решить поменять порядок выполнения операций, если, например, сочтет, что такая последовательность выполнится быстрее. Эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кеш, а результат второй операции попадает непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти, может сначала увидеть результат второй операции и только потом первой, когда все регистры или кеши синхронизируются с основной памятью.

Также регулируется набором правил «happens-before»: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не volatile переменных.

https://habr.com/ru/company/golovachcourses/blog/221133/

SQL

Что такое DDL? Какие операции в него входят? Рассказать про них

Операторы определения данных (Data Definition Language, DDL):

- **CREATE** создает объект БД (базу, таблицу, представление, пользователя и т. д.);
- **ALTER** изменяет объект;
- **DROP** удаляет объект;
- **TRUNCATE** удаляет таблицу и создает ее пустую заново, но если в таблице были foreigh key, то создать таблицу не получится, rollback после TRUNCATE невозможен.

Что такое DML? Какие операции в него входят? Рассказать про них

Операторы манипуляции данными (Data Manipulation Language, DML):

- **SELECT** выбирает данные, удовлетворяющие заданным условиям;
- **INSERT** добавляет новые данные;
- **UPDATE** изменяет существующие данные;
- **DELETE** удаляет данные при выполнении условия **WHERE**;

Что такое TCL? Какие операции в него входят? Рассказать про них

Операторы управления транзакциями (Transaction Control Language, TCL):

- **BEGIN** служит для определения начала транзакции;
- **COMMIT** применяет транзакцию;
- **ROLLBACK** откатывает все изменения, сделанные в контексте текущей транзакции;
- **SAVEPOINT** разбивает транзакцию на более мелкие.

SetTransaction –

Что такое DCL? Какие операции в него входят? Рассказать про них

Операторы определения доступа к данным (Data Control Language, DCL):

- **GRANT** предоставляет пользователю (группе) разрешения на определенные операции с объектом;
- **REVOKE** отзывает ранее выданные разрешения;
- **DENY** задает запрет, имеющий приоритет над разрешением.

Нюансы работы с NULL в SQL. Как проверить поле на NULL?

NULL – **специальное значение** (псевдозначение), которое может быть записано в поле таблицы базы данных. NULL соответствует понятию «пустое поле», то есть «поле, не содержащее никакого значения».

NULL **означает отсутствие**, неизвестность **информации**. Значение NULL не является значением в полном смысле слова: по определению оно означает отсутствие значения и не принадлежит ни одному типу данных. Поэтому NULL не равно ни логическому значению

FALSE, ни пустой строке, ни 0. **При сравнении NULL с любым значением будет получен результат NULL**, а не FALSE и не 0. Более того, **NULL не равно NULL**!

Команды: **IS NULL**, **IS NOT NULL**.

Виды Join'ов?

JOIN – оператор языка SQL, который является реализацией операции соединения реляционной алгебры. Предназначен для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор.

Особенностями операции соединения являются следующее:

- в схему таблицы-результата входят столбцы обеих исходных таблиц (таблиц-операндов), то есть схема результата является «сцеплением» схем операндов;
- каждая строка таблицы-результата является «сцеплением» строки из одной таблицы-операнда со строкой второй таблицы-операнда;
- при необходимости соединения не двух, а нескольких таблиц, операция соединения применяется несколько раз (последовательно).

SELECT

field_name [,... n]

FROM

Table1

{INNER | {LEFT | RIGHT | FULL} OUTER | CROSS } JOIN

Table2

{ON <condition> | USING (field_name [,... n])}

Какие существуют типы JOIN?

- **(INNER) JOIN** Результатом объединения таблиц являются записи, общие для левой и правой таблиц. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.
- **LEFT (OUTER) JOIN**. Производит выбор всех записей первой таблицы и соответствующих им записей второй таблицы. Если записи во второй таблице не найдены, то вместо них подставляется пустой результат (NULL). Порядок таблиц для оператора важен, поскольку оператор не является симметричным.
- **RIGHT (OUTER) JOIN** с операндами, расставленными в обратном порядке. Порядок таблиц для оператора важен, поскольку оператор не является симметричным.
- **FULL (OUTER) JOIN**. Результатом объединения таблиц являются все записи, которые присутствуют в таблицах. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.
- **CROSS JOIN** (декартово произведение) При выборе каждая строка одной таблицы объединяется с каждой строкой второй таблицы, давая тем самым все возможные сочетания строк двух таблиц. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.

Что лучше использовать join или подзапросы? Почему?

Обычно лучше использовать JOIN, поскольку в большинстве случаев он более понятен и лучше оптимизируется СУБД (но 100% этого гарантировать нельзя). Так же JOIN имеет заметное преимущество над подзапросами в случае, когда список выбора SELECT содержит столбцы более чем из одной таблицы.

Подзапросы лучше использовать в случаях, когда нужно вычислять агрегатные значения и использовать их для сравнений во внешних запросах.

Что делает UNION?

В языке SQL ключевое слово UNION применяется для объединения результатов двух SQL-запросов в единую таблицу, состоящую из схожих записей. Оба запроса должны возвращать одинаковое число столбцов и совместимые типы данных в соответствующих столбцах. Необходимо отметить, что UNION сам по себе не гарантирует порядок записей. Записи из второго запроса могут оказаться в начале, в конце или вообще перемешаться с записями из первого запроса. В случаях, когда требуется определенный порядок, необходимо использовать ORDER BY.

Разница между UNION и UNION ALL заключается в том, что UNION будет пропускать дубликаты записей, тогда как UNION ALL будет включать дубликаты записей.

Чем WHERE отличается от HAVING (ответа про то, что используются в разных частях запроса недостаточно)?

WHERE нельзя использовать с агрегатными функциями, HAVING можно (предикаты тоже).

В HAVING можно использовать псевдонимы, только если они используются для наименования результата агрегатной функции, в WHERE можно всегда.

HAVING стоит после GROUP BY, но может использоваться и без него. При отсутствии предложения GROUP BY агрегатные функции применяются ко всему выходному набору строк запроса, т. е. в результате получим всего одну строку, если выходной набор не пуст.

Что такое ORDER BY?

ORDER BY упорядочивает вывод запроса согласно значениям в том или ином количестве выбранных столбцов. Многочисленные столбцы упорядочиваются один внутри другого. Возможно определять возрастание **ASC** или убывание **DESC** для каждого столбца. По умолчанию установлено **возрастание**.

Что такое GROUP BY?

GROUP BY используется для агрегации записей результата по заданным атрибутам.

Создает отдельную группу для всех возможных значений (включая значение NULL).

При использовании GROUP BY все значения NULL считаются равными.

Что такое DISTINCT?

DISTINCT указывает, что для вычислений используются только уникальные значения столбца.

Что такое LIMIT?

Ограничивает выборку заданным числом.

Что такое EXISTS?

EXISTS берет подзапрос, как аргумент, и оценивает его как TRUE, если подзапрос возвращает какие-либо записи, и FALSE, если нет.

Расскажите про операторы IN, BETWEEN, LIKE

- **IN** определяет набор значений.

*SELECT * FROM Persons WHERE name IN ('Ivan','Petr','Pavel');*

- **BETWEEN** определяет диапазон значений. В отличие от IN, BETWEEN чувствителен к порядку, и первое значение в предложении должно быть первым по алфавитному или числовому порядку.

*SELECT * FROM Persons WHERE age BETWEEN 20 AND 25;*

- **LIKE** применим только к полям типа CHAR или VARCHAR, с которыми он используется чтобы находить подстроки. В качестве условия используются символы шаблонизации (wildkards) – специальные символы, которые могут соответствовать чему-нибудь:
  - _ замещает любой одиночный символ. Например, 'b_t' будет соответствовать словам 'bat' или 'bit', но не будет соответствовать 'brat'.
  - % замещает последовательность любого числа символов. Например '%p%t' будет соответствовать словам 'put', 'posit', или 'opt', но не 'spite'.

*SELECT * FROM UNIVERSITY WHERE NAME LIKE '%o';*

Что делает оператор MERGE? Какие у него есть ограничения?

MERGE позволяет осуществить слияние данных одной таблицы с данными другой таблицы. При слиянии таблиц проверяется условие, и если оно истинно, то выполняется UPDATE, а если нет – INSERT. При этом изменять поля таблицы в секции UPDATE, по которым идет связывание двух таблиц, нельзя.

MERGE Ships AS t -- таблица, которая будет меняться

USING (SELECT запрос) AS s ON (t.name = s.ship) – условие слияния

THEN UPDATE SET t.launched = s.year – обновление

WHEN NOT MATCHED – если условие не выполняется

THEN INSERT VALUES(s.ship, s.year) – вставка

Какие агрегатные функции вы знаете?

Агрегатных функции – функции, которые берут группы значений и сводят их к одиночному значению.

Несколько агрегатных функций:

- **COUNT** производит подсчет записей, удовлетворяющих условию запроса;
- **CONCUT** соединяет строки;
- **SUM** вычисляет арифметическую сумму всех значений колонки;
- **AVG** вычисляет среднее арифметическое всех значений;

- **MAX** определяет наибольшее из всех выбранных значений;
- **MIN** определяет наименьшее из всех выбранных значений.

Что такое ограничения (constraints)? Какие вы знаете?

Ограничения – это ключевые слова, которые помогают установить правила размещения данных в базе. Используются при создании БД.

NOT NULL указывает, что значение не может быть пустым.

UNIQUE обеспечивает отсутствие дубликатов.

PRIMARY KEY – комбинация NOT NULL и UNIQUE. Помечает каждую запись в базе данных уникальным значением.

CHECK проверяет, вписывается ли значение в заданный диапазон (s_id int CHECK(s_id > 0)).

FOREIGN KEY создает связь между двумя таблицами и защищает от действий, которые могут нарушить связи между таблицами. FOREIGN KEY в одной таблице указывает на PRIMARY KEY в другой.

DEFAULT устанавливает значение по умолчанию, если значения не предоставлено (name VARCHAR(20) DEFAULT 'noname').

Какие отличия между PRIMARY и UNIQUE?

По умолчанию PRIMARY создает кластерный индекс на столбце, а UNIQUE – некластерный. PRIMARY не разрешает NULL записей, в то время как UNIQUE разрешает одну (а в некоторых СУБД несколько) NULL запись.

Таблица может иметь один PRIMARY KEY и много UNIQUE.

Может ли значение в столбце, на который наложено ограничение FOREIGN KEY, равняться NULL?

Может, если на данный столбец не наложено ограничение NOT NULL.

Что такое суррогатные ключи?

Суррогатный ключ – это дополнительное служебное поле, автоматически добавленное к уже имеющимся информационным полям таблицы, предназначение которого – служить первичным ключом.

Что такое индексы? Какие они бывают?

Индексы относятся к методу настройки производительности, позволяющему быстрее извлекать записи из таблицы. Индекс создает структуру для индексируемого поля. Необходимо просто добавить указатель индекса в таблицу.

Есть три типа индексов:

Уникальный индекс (Unique Index): этот индекс не позволяет полю иметь повторяющиеся значения. Если первичный ключ определен, уникальный индекс применен автоматически.

Кластеризованный индекс (Clustered Index): сортирует и хранит строки данных в таблицах или представлениях на основе их ключевых значений. Это ускоряет операции чтения из БД.

Некластеризованный индекс (Non-Clustered Index): внутри таблицы есть упорядоченный список, содержащий значения ключа некластеризованного индекса и указатель на строку данных, содержащую значение ключа. Каждый новый индекс увеличивает время, необходимое для создания новых записей из-за упорядочивания. Каждая таблица может иметь много некластеризованных индексов.

Как создать индекс?

- с помощью выражения **CREATE INDEX**:

CREATE INDEX index_name ON table_name (column_name)

- указав ограничение целостности в виде уникального UNIQUE или первичного PRIMARY ключа в операторе создания таблицы CREATE TABLE.

Имеет ли смысл индексировать данные, имеющие небольшое количество возможных значений?

Примерное правило, которым можно руководствоваться при создании индекса: если объем информации (в байтах) НЕ удовлетворяющей условию выборки меньше, чем размер индекса (в байтах) по данному условию выборки, то в общем случае оптимизация приведет к замедлению выборки.

Когда полное сканирование набора данных выгоднее доступа по индексу?

Полное сканирование производится многоблочным чтением. Сканирование по индексу – одноблочным. При доступе по индексу сначала идет сканирование самого индекса, а затем чтение блоков из набора данных. Число блоков, которые надо при этом прочитать из набора зависит от фактора кластеризации. Если суммарная стоимость всех необходимых одноблочных чтений больше стоимости полного сканирования многоблочным чтением, то полное сканирование выгоднее и оно выбирается оптимизатором.

Таким образом, полное сканирование выбирается при слабой селективности предикатов запроса и/или слабой кластеризации данных, либо в случае очень маленьких наборов данных.

Чем TRUNCATE отличается от DELETE?

DELETE – оператор DML, удаляет записи из таблицы, которые удовлетворяют условиям WHERE. Медленнее, чем TRUNCATE. Есть возможность восстановить данные.

TRUNCATE – DDL оператор, удаляет все строки из таблицы. Нет возможность восстановить данные – сделать ROLLBACK.

Что такое хранимые процедуры? Для чего они нужны?

Хранимая процедура – объект базы данных, представляющий собой набор SQL-инструкций, который хранится на сервере.

Хранимые процедуры очень похожи на обыкновенные методы языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам.

В хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения.

Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных.

В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным) и в дальнейшем ее обработка осуществляется быстрее.

Что такое «триггер»?

Триггер (trigger) – это хранимая процедура особого типа, исполнение которой обусловлено действием по модификации данных: добавлением, удалением или изменением данных в заданной таблице реляционной базы данных. Триггер запускается сервером автоматически и все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера.

Момент запуска триггера определяется с помощью ключевых слов **BEFORE** (триггер запускается до выполнения связанного с ним события) или **AFTER** (после события).

Что такое представления (VIEW)? Для чего они нужны?

View – виртуальная таблица, представляющая данные одной или более таблиц альтернативным образом.

В действительности представление – всего лишь результат выполнения оператора SELECT, который хранится в структуре памяти, напоминающей SQL таблицу. Они работают в запросах и операторах DML точно так же, как и основные таблицы, но не содержат никаких собственных данных. Представления значительно расширяют возможности управления данными. Это способ дать публичный доступ к некоторой (но не всей) информации в таблице.

Представления могут основываться как на таблицах, так и на других представлениях, т. е. могут быть вложенными (до 32 уровней вложенности).

Что такое временные таблицы? Для чего они нужны?

Подобные таблицы удобны для каких-то временных промежуточных выборок из нескольких таблиц.

Создание временной таблицы начинается со знака решетки #. Если используется один знак #, то создается локальная таблица, которая доступна в течение текущей сессии. Ели используются два знака ##, то создается глобальная временная таблица. В отличие от локальной глобальная временная таблица доступна всем открытым сессиям базы данных.

CREATE TABLE #ProductSummary

(ProdId INT IDENTITY,

ProdName NVARCHAR(20),

Price MONEY)

Что такое транзакции? Расскажите про принципы ACID

Транзакция – это воздействие на базу данных, переводящее ее из одного целостного состояния в другое и выражаемое в изменении данных, хранящихся в базе данных.

ACID-принципы транзакций:

- **Атомарность** (atomicity) гарантирует, что транзакция будет полностью выполнена или потерпит неудачу, где транзакция представляет одну логическую операцию данных. Это означает, что при сбое одной части любой транзакции происходит сбой всей транзакции и состояние базы данных остается неизменным.
- **Согласованность** (consistency). Транзакция, достигающая своего завершения и фиксирующая свои результаты, сохраняет согласованность базы данных.
- **Изолированность** (isolation). Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на ее результат.
- **Долговечность** (durability). Независимо от проблем (к примеру, потеря питания, сбой или ошибки любого рода) изменения, сделанные успешно завершенной транзакцией, должны остаться сохраненными после возвращения системы в работу.

Расскажите про уровни изолированности транзакций

«Грязное» чтение (Dirty Read): транзакция A производит запись. Между тем транзакция B считывает ту же самую запись до завершения транзакции A. Позже транзакция A решает откатится, и теперь у нас есть изменения в транзакции B, которые несовместимы. Это грязное чтение. Транзакция B работала на уровне изоляции READ_UNCOMMITTED, поэтому она могла считывать изменения, внесенные транзакцией A до того, как транзакция завершилась.

Неповторяющееся чтение (Non-Repeatable Read): транзакция A считывает некоторые записи. Затем транзакция B записывает эту запись и фиксирует ее. Позже транзакция A снова считывает эту же запись и может получить разные значения, поскольку транзакция B вносила изменения в эту запись и фиксировала их. Это неповторяющееся чтение.

Фантомные чтение (Phantom Read): транзакция A читает ряд записей. Между тем транзакция B вставляет новую запись в этот же ряд, что и транзакция A. Позднее транзакция A снова считывает тот же диапазон и также получит запись, которую только что вставила транзакция B. Это фантомное чтение: транзакция извлекала ряд записей несколько раз из базы данных и получала разные результирующие наборы (содержащие фантомные записи).

Что такое нормализация и денормализация? Расскажите про 3 нормальные формы

Нормализация – это процесс преобразования отношений базы данных к виду, отвечающему нормальным формам (пошаговый, обратимый процесс приведения данных в более простую и логичную структуру).

Целью является уменьшение потенциальной противоречивости хранимой в базе данных информации.

Денормализация базы данных – это процесс обратный от нормализации. Эта техника добавляет избыточные данные в таблицу, учитывая частые запросы к базе данных, которые объединяют данные из разных таблиц в одну таблицу. Необходимо для повышения производительности и скорости извлечения данных за счет увеличения избыточности данных.

Каждая нормальная форма включает в себя предыдущую. Типы форм:

- **Первая** нормальная форма (1NF) – значения всех полей атомарны (неделимы), нет множества значений в одном поле.

Требование первой нормальной формы (1NF) очень простое и оно заключается в том, чтобы таблицы соответствовали реляционной модели данных и соблюдали определённые реляционные принципы.

Таким образом, чтобы база данных находилась в 1 нормальной форме, необходимо чтобы ее таблицы соблюдали следующие реляционные принципы:

  - В таблице не должно быть дублирующих строк
  - В каждой ячейке таблицы хранится атомарное значение (одно не составное значение)
  - В столбце хранятся данные одного типа
  - Отсутствуют массивы и списки в любом виде

- **Вторая** нормальная форма (2NF) – все неключевые поля зависят только от ключа целиком, а не от какой-то его части.

Чтобы база данных находилась во второй нормальной форме (2NF), необходимо чтобы ее таблицы удовлетворяли следующим требованиям:

  - Таблица должна находиться в первой нормальной форме
  - Таблица должна иметь ключ
  - Все неключевые столбцы таблицы должны зависеть от полного ключа (*в случае если он составной*)

- **Третья** нормальная форма (3NF) – все неключевые поля не зависят друг от друга.

Требование третьей нормальной формы (3NF) заключается в том, чтобы в таблицах отсутствовала транзитивная зависимость.

Транзитивная зависимость – это когда неключевые столбцы зависят от значений других неключевых столбцов.

- **Нормальная форма Бойса-Кодда**, усиленная 3 нормальная форма (BCNF) – когда каждая ее нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ.

Требования нормальной формы Бойса-Кодда следующие:

  - Таблица должна находиться в третьей нормальной форме. Здесь все как обычно, т.е. как и у всех остальных нормальных форм, первое требование заключается в том, чтобы таблица находилась в предыдущей нормальной форме, в данном случае в третьей нормальной форме;
  - Ключевые атрибуты составного ключа не должны зависеть от неключевых атрибутов.

- **Четвертая** нормальная форма (4NF) – не содержатся независимые группы полей, между которыми существует отношение «многие-ко-многим».

Требование четвертой нормальной формы (4NF) заключается в том, чтобы в таблицах отсутствовали нетривиальные многозначные зависимости.

В таблицах многозначная зависимость выглядит следующим образом.

Начнем с того, что таблица должна иметь как минимум три столбца, допустим A, B и C, при этом B и C между собой никак не связаны и не зависят друг от друга, но по отдельности зависят от A, и для каждого значения A есть множество значений B, а также множество значений C.

В данном случае многозначная зависимость обозначается вот так:

A —> B

A —> C

- **Пятая** нормальная форма (5NF) – каждая нетривиальная зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения.
- **Доменно-ключевая** нормальная форма (DKNF) – каждое наложенное на нее ограничение является логическим следствием ограничений доменов и ограничений ключей, наложенных на данное отношение.

Ограничение домена – это ограничение, предписывающее использование для определенного атрибута значений только из некоторого заданного домена (набора значений).

Ограничение ключа – это ограничение, утверждающее, что некоторый атрибут или комбинация атрибутов представляет собой потенциальный ключ.

Таким образом, требование доменно-ключевой нормальной формы заключается в том, чтобы каждое наложенное ограничение на таблицу являлось логическим следствием ограничений доменов и ограничений ключей, которые накладываются на данную таблицу.

- **Шестая** нормальная форма (6NF) – удовлетворяет всем нетривиальным зависимостям соединения, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Введена как обобщение пятой нормальной формы для хронологической базы данных.

Хронологическая база данных – это база, которая может хранить не только текущие данные, но и исторические данные, т. е. данные, относящиеся к прошлым периодам времени. Однако такая база может хранить и данные, относящиеся к будущим периодам времени.

Что такое TIMESTAMP?

DATETIME предназначен для хранения целого числа: YYYYMMDDHHMMSS. И это время не зависит от временной зоны настроенной на сервере. Размер 8 байт.

TIMESTAMP хранит значение равное количеству секунд, прошедших с полуночи 1 января 1970 года по усредненному времени Гринвича. Тогда была создана Unix. При получении из базы отображается с учетом часового пояса. Размер 4 байта.

Шардирование БД

При большом количестве данных запросы начинают долго выполняться, и сервер начинает не справляться с нагрузкой. Одно из решений – масштабирование базы данных. Например, шардинг или репликация.

Шардинг бывает вертикальным (партицирование) и горизонтальным.

У нас есть большая таблица, например, с пользователями. Партицирование – это когда мы одну большую таблицу разделяем на много маленьких по какому-либо принципу.

Единственное отличие горизонтального масштабирования от вертикального в том, что горизонтальное будет разносить данные по разным инстансам в других базах.

```
CREATE TABLE news (
    id bigint not null,
    category_id int not null,
    author character varying not null,
    rate int not null,
    title character varying
)
```

Есть таблица news**,** в которой есть идентификатор, есть категория, в которой эта новость расположена, есть автор новости.

Нужно сделать 2 действия над табличкой

1. Поставить у нашего шарда, например, news_1, то, что она будет наследоваться от news.

Наследованная таблица будет иметь все колонки родителя, а также она может иметь свои колонки, которые мы дополнительно туда добавим. Там не будет ограничений, индексов и триггеров от родителя. Это важно.

2. Поставить ограничения. Это будет проверка, что в эту таблицу будут попадать данные только с нужным признаком.

```
CREATE TABLE news_1 (
    CHECK ( category_id = 1 )
) INHERITS (news)
```

Т. е. только записи с category_id=1 будут попадать в эту таблицу.

На базовую таблицу надо добавить правило. Когда будем работать с таблицей news, вставка на запись с category_id = 1 должна попасть именно в партицию news_1. Правило называем, как хотим.

```
CREATE RULE news_insert_to_1 AS ON INSERT TO news
WHERE ( category_id = 1 )
DO INSTEAD INSERT INTO news_1 VALUES (NEW.*)
```

EXPLAIN

Когда выполняете какой-нибудь запрос, оптимизатор запросов MySQL пытается придумать оптимальный план выполнения этого запроса. Можно посмотреть этот план, используя запрос с ключевым словом EXPLAIN перед оператором SELECT.

*EXPLAIN SELECT * FROM categories*

После EXPLAIN в запросе можно использовать ключевое слово EXTENDED, и MySQL покажет дополнительную информацию о том, как выполняется запрос. Чтобы увидеть эту информацию, нужно сразу после запроса с EXTENDED выполнить запрос SHOW WARNINGS.

*EXPLAIN **EXTENDED** SELECT City.Name FROM City*

Затем

SHOW WARNINGS

Как сделать запрос из двух баз?

Если в запросе таблица указывается с именем базы данных database1.table1, то таблица выбирается из database1, если просто table1, то из активной базы данных.

Надо, чтобы базы были на одном сервере.

SELECT t1., t2.**

FROM database1.table1 AS t1

INNER JOIN database2.table2 AS t2 ON t1.field1 = t2.field1

Что быстрее убирает дубликаты: distinct или group by?

Если нужны уникальные значения – DISTINCT.

Если нужно группировать значения – GROUP BY.

Если задача заключается именно в поиске дубликатов – **GROUP BY будет лучше.**

Hibernate

Что такое ORM? Что такое JPA? Что такое Hibernate?

ORM (Object Relational Mapping) – это концепция преобразования данных из объектно-ориентированного языка в реляционные БД и наоборот.

JPA (Java Persistence API) – это стандартная для Java спецификация, описывающая принципы ORM. JPA не умеет работать с объектами, а только определяет правила, как должен действовать каждый провайдер (Hibernate, EclipseLink), реализующий стандарт JPA.

JPA определяет правила, по которым должны описываться метаданные отображения и как должны работать провайдеры. Каждый провайдер обязан реализовывать все из JPA, определяя стандартное получение, сохранение, управление объектами. Можно добавлять свои классы и интерфейсы.

Гибкость – код написанный с использование классов и интерфейсов JPA, позволяет гибко менять одного провайдера на другого, но если использовать классы, аннотации и интерфейсы из конкретного провайдера, то это работать не будет.

JDO входит в JPA, NoSQL.

Hibernate – библиотека, являющаяся реализацией JPA-спецификации, в которой можно использовать не только стандартные API-интерфейсы JPA, но и реализовать свои классы и интерфейсы.

Важные интерфейсы Hibernate:

Session – обеспечивает физическое соединение между приложением и БД. Основная функция – предлагать DML-операции для экземпляров сущностей.

SessionFactory – это фабрика для объектов Session. Обычно создается во время запуска приложения и сохраняется для последующего использования. Является потокобезопасным объектом и используется всеми потоками приложения.

Transaction – однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC-транзакций. Session может занимать несколько Transaction в определенных случаях, является необязательным API.

Query – интерфейс позволяет выполнять запросы к БД. Запросы написаны на HQL или на SQL.

Что такое EntityManager? Какие функции он выполняет?

EntityManager – интерфейс JPA, который описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA.

Основные операции:

1. **Операции над Entity:** persist (добавление Entity), merge (обновление), remove (удаление), refresh (обновление данных), detach (удаление из управление JPA), lock (блокирование Entity от изменений в других thread).
2. **Получение данных**: find (поиск и получение Entity), createQuery, createNamedQuery, createNativeQuery, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery.

3. **Получение других сущностей JPA**: getTransaction, getEntityManagerFactory, getCriteriaBuilder, getMetamodel, getDelegate.
4. **Работа с EntityGraph**: createEntityGraph, getEntityGraph.
5. **Общие операции** над EntityManager или всеми Entities: close, clear, isOpen, getProperties, setProperty.

Объекты EntityManager не являются потокобезопасными. Это означает, что каждый поток должен получить свой экземпляр EntityManager, поработать с ним и закрыть его в конце.

Каким условиям должен удовлетворять класс, чтобы являться Entity?

Entity – это легковесный хранимый объект бизнес логики. Основная программная сущность – это **entity-класс**, который может использовать дополнительные классы, которые могут использоваться как вспомогательные классы или для сохранения состояния entity.

Требования к entity-классу:

- должен быть помечен аннотацией Entity или описан в XML-файле;
- должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами) – при получении данных из БД и формировании из них объекта сущности Hibernate должен создать этот объект сущности$
- должен быть классом верхнего уровня (top-level class);
- не может быть enum или интерфейсом;
- не может быть финальным классом (final class);
- не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables);
- если объект entity-класса будет передаваться по значению как отдельный объект (detached object), например, через удаленный интерфейс (through a remote interface), он должен реализовывать интерфейс Serializable;
- поля entity-класса должны быть напрямую доступны только методам самого entity-класса и не должны быть напрямую доступны другим классам, использующим этот entity. Такие классы должны обращаться только к методам (getter/setter методам или другим методам бизнес-логики в entity-классе);
- должен содержать первичный ключ, то есть атрибут или группу атрибутов, которые уникально определяют запись этого entity-класса в базе данных.

Может ли абстрактный класс быть Entity?

Может, при этом он сохраняет все свойства Entity, за исключением того, что его нельзя непосредственно инициализировать.

Может ли entity-класс наследоваться от не entity-классов (non-entity classes)?

Может.

Может ли entity-класс наследоваться от других entity-классов?

Может.

Может ли НЕ entity-класс наследоваться от entity-класса?

Может.

Что такое встраиваемый (embeddable) класс? Какие требования JPA устанавливает к встраиваемым (embeddable) классам?

Embeddable-класс – это класс, который не используется сам по себе, а является частью одного или нескольких entity-классов. Entity-класс может содержать как одиночные встраиваемые классы, так и коллекции таких классов. Также такие классы могут быть использованы как ключи или значения map. Во время выполнения каждый встраиваемый класс принадлежит только одному объекту entity-класса и не может быть использован для передачи данных между объектами entity-классов (то есть такой класс не является общей структурой данных для разных объектов). В целом, такой класс служит для того, чтобы выносить определение общих атрибутов для нескольких entity.

Такие классы должны удовлетворять тем же правилам, что entity-классы, за исключением того, что они не обязаны содержать первичный ключ и быть отмечены аннотацией Entity.

Embeddable-класс должен быть помечен аннотацией @Embeddable или описан в XML-файле конфигурации JPA. А поле этого класса в Entity аннотацией @Embedded.

Embeddable-класс может содержать другой встраиваемый класс.

Встраиваемый класс может содержать связи с другими Entity или коллекциями Entity, если такой класс не используется как первичный ключ или ключ map'ы.

Что такое Mapped Superclass?

Mapped Superclass – это класс, от которого наследуются Entity, он может содержать анотации JPA, однако сам такой класс не является Entity, ему не обязательно выполнять все требования, установленные для Entity (например, он может не содержать первичного ключа). Такой класс не может использоваться в операциях EntityManager или Query. Такой класс должен быть отмечен аннотацией MappedSuperclass или описан в xm- файле.

Создание такого класса-предка оправдано тем, что заранее определяется ряд свойств и методов в сущностях. Использование такого подхода позволило сократить количество кода.

Какие три типа стратегий наследования мапинга (Inheritance Mapping Strategies) описаны в JPA?

Inheritance Mapping Strategies описывает как JPA будет работать с классами-наследниками Entity:

1. **Одна таблица на всю иерархию классов (SINGLE_TABLE)** – все entity со всеми наследниками записываются в одну таблицу, для идентификации типа entity определяется специальная колонка «discriminator column». Например, есть entity Animals с классами-потомками Cats и Dogs. При такой стратегии все entity записываются в таблицу Animals, но при этом имеют дополнительную колонку animalType, в которую соответственно пишется значение «cat» или «dog». Минусом является то, что в общей таблице будут созданы все поля, уникальные для каждого из классов-потомков, которые будут пусты для всех других классов-потомков. Например, в таблице animals окажется и скорость лазанья по дереву от cats, и может ли пес приносить тапки от dogs, которые будут всегда иметь null для dog и cat соответственно.

Нельзя делать constraints notNull, но можно использовать триггеры.

2. **Стратегия «соединения» (JOINED_TABLE)** – в этой стратегии каждый класс entity сохраняет данные в свою таблицу, но только уникальные поля (не унаследованные от классов-предков) и первичный ключ, а все унаследованные колонки записываются в таблицы класса-предка, дополнительно устанавливается связь (relationships) между этими таблицами, например, в случае классов Animals будут три таблицы: animals, cats, dogs. Причем в cats будет записан только ключ и скорость лазанья, в dogs – ключ и умеет ли пес приносить палку, а в animals все остальные данные cats и dogs с ссылкой на соответствующие таблицы. Минусом является потеря производительности от объединения таблиц (join) для любых операций.

3. **Таблица для каждого класса (TABLE_PER_CLASS)** – каждый отдельный класс-наследник имеет свою таблицу, т. е. для cats и dogs все данные будут записываться просто в таблицы cats и dogs как если бы они вообще не имели общего суперкласса. Минусом является плохая поддержка полиморфизма (polymorphic relationships) и то, что для выборки всех классов иерархии потребуются большое количество отдельных sql-запросов или использование UNION-запроса.

Для задания стратегии наследования используется аннотация Inheritance (или соответствующие блоки).

Как мапятся Enum'ы?

@Enumerated(EnumType.STRING) означает, что в базе будут храниться имена Enum.

@Enumerated(EnumType.ORDINAL) – в базе будут храниться порядковые номера Enum.

Другой вариант – можно смапить enum в БД и обратно в методах с аннотациями @PostLoad и @PrePersist. @EntityListener над классом Entity, где указать класс, в котором создать два метода, помеченных этими аннотациями.

Идея в том, чтобы в сущности иметь не только поле с Enum, но и вспомогательное поле. Поле с Enum аннотируем @Transient, а в БД будет храниться значение из вспомогательного поля.

В JPA с версии 2.1 можно использовать Converter для конвертации Enum’а в некое его значение для сохранения в БД и получения из БД. Нужно лишь создать новый класс, который реализует javax.persistence.AttributeConverter и аннотировать его с помощью @Converter и поле в сущности аннотацией @Convert.

Как мапятся даты (до Java 8 и после)?

Аннотация @Temporal до Java 8, в которой надо было указать, какой тип даты хотим использовать.

В Java 8 и далее аннотацию ставить не нужно.

Как «смапить» коллекцию примитивов?

@ElementCollection

@OrderBy

Если у сущности есть поле с коллекцией, то обычно ставят над ним аннотации @OneToMany либо @ManyToMany. Но данные аннотации применяются, когда это коллекция других сущностей (entities). Если у сущности коллекция не других сущностей, а базовых или встраиваемых (embeddable) типов, то для этих случаев в JPA имеется специальная аннотация @ElementCollection, которая указывается в классе сущности над полем

коллекции. Все записи коллекции хранятся в отдельной таблице, то есть в итоге получаем две таблицы: одну для сущности, вторую для коллекции элементов.

При добавлении новой строки в коллекцию она полностью очищается и заполняется заново, так как у элементов нет id. Можно решить с помощью @OrderColumn.

@CollectionTable позволяет редактировать таблицу с коллекцией.

Какие есть виды связей?

Существуют 4 типа связей:

1. **OneToOne** – когда один экземпляр Entity может быть связан не больше чем с одним экземпляром другого Entity.

Необходимо ставить forieghnKey на родительскую таблицу и аннотацию JoinColomun, в атрибуте name объяснить, к какой колонке ссылаться на родительской сущности.

Что стоит в поле, где все связано? Стоит тип другой сущности.

2. **OneToMany** – когда один экземпляр Entity может быть связан с несколькими экземплярами других Entity. Когда одна сущность может ссылаться ко многим сущностям.

Храним коллекцию.

3. **ManyToOne** – обратная связь для OneToMany. Несколько экземпляров Entity могут быть связаны с одним экземпляром другого Entity. Несколько машин может быть у нескольких юзеров.

Одна сущность хранится.

4. **ManyToMany** – экземпляры Entity могут быть связаны с несколькими экземплярами друг друга. Каждый из двух Entity может быть по несколько других Entity. Много сущностей могут относиться к многим сущностями.

Сводная таблица с айдишниками, коллекции коллекций хранятся.

Каждую из которых можно разделить ещё на два вида:

1. Bidirectional с использованием @MappedBy на стороне, где указывается @OneToMany

2. Unidirectional.

Bidirectional – ссылка на связь устанавливается у всех Entity, то есть в случае OneToOne A-B в Entity A есть ссылка на Entity B, в Entity B есть ссылка на Entity A. Entity A считается владельцем этой связи (это важно для случаев каскадного удаления данных, тогда при удалении A также будет удалено B, но не наоборот).

Undirectional – ссылка на связь устанавливается только с одной стороны, то есть в случае OneToOne A-B только у Entity A будет ссылка на Entity B, у Entity B ссылки на A не будет.

Что такое владелец связи?

В отношениях между двумя сущностями всегда есть одна владеющая сторона, а зависимой может и не быть, если это однонаправленные отношения.

По сути, у кого есть внешний ключ на другую сущность, тот и владелец связи. То есть, если в таблице одной сущности есть колонка, содержащая внешние ключи от другой сущности, то первая сущность признается владельцем связи, вторая сущность – зависимой.

В однонаправленных отношениях сторона, которая имеет поле с типом другой сущности, является владельцем этой связи по умолчанию.

Что такое каскады?

Каскадирование – это какое-то действие с целевой Entity, то же самое действие будет применено к связанной Entity.

JPA CascadeType:

- **ALL** гарантирует, что все персистентные события, которые происходят на родительском объекте, будут переданы дочернему объекту;
- **PERSIST** означает, что операции save() или persist() каскадно передаются связанным объектам;
- **MERGE** означает, что связанные entity объединяются, когда объединяется entity-владелец;
- **REMOVE** удаляет все entity, связанные с удаляемой entity;
- **DETACH** отключает все связанные entity, если происходит «ручное отключение»;
- **REFRESH** повторно считывает значение данного экземпляра и связанных сущностей из базы данных при вызове refresh().

Разница между PERSIST и MERGE?

persist(entity) следует использовать с новыми объектами, чтобы добавить их в БД (если объект уже существует в БД, будет выброшено исключение EntityExistsException).

Если использовать **merge(entity)**, то сущность, которая уже управляется в контексте персистентности, будет заменена новой сущностью (обновленной), и копия этой обновленной сущности вернется обратно. Рекомендуется использовать для уже сохраненных сущностей.

Какие два типа fetch-стратегии в JPA вы знаете?

1. **LAZY** – Hibernate может загружать данные не сразу, а при первом обращении к ним, но так как это необязательное требование, то Hibernate имеет право изменить это поведение и загружать их сразу. Это поведение по умолчанию для полей, аннотированных @OneToMany, @ManyToMany и @ElementCollection. В объект загружается прокси lazy-поля. Если там стоит коллекция, то это будет коллекция Hibernate, именуемая типом коллекции bag().

Подгрузка должна происходить в одной транзакции или пока не закроем ЕнтитуМенеджер.

Если обратимся за персистетнымКонтекстом, то LazyEnitialization.

Если используем Бэк, то он использует Лист и Сет и т. д.

2. **EAGER** – данные поля будут загружены немедленно. Это поведение по умолчанию для полей, аннотированных @Basic, @ManyToOne и @OneToOne.

Все, что заканчивается на One – Eager, Many – Lazy.

Для кого можем использовать ManyToOne? Для владельца связи.

Какие четыре статуса жизненного цикла Entity-объекта (Entity Instance's Life Cycle) вы можете перечислить?

- **transient (new)** – свежесозданная оператором new() сущность не имеет связи с базой данных, не имеет данных в базе данных и не имеет сгенерированных первичных ключей и не имеет контекста Персистентности. При сохранении переходит в managed.
- **managed** – объект уже создан и получает первичный ключ, управляется контекстом персистентности.(сохранен в БД, имеет primary key), переходит под управляется JPA, если вызываем detached, то полностью отвязываем от контекста.
- **detached** – не управляется JPA, но может существовать в БД, объект создан, но не управляется JPA. В этом состоянии сущность не связана со своим контекстом (отделена от него) и нет экземпляра Session, который бы ей управлял.
- **removed** – объект создан, управляется JPA, будет удален из БД, при commit-е транзакции статус станет опять detached.

Как влияет операция persist на Entity-объекты каждого из четырех статусов?

- new → managed, объект будет сохранен в базу при commit-е транзакции или в результате flush-операций;
- managed → операция игнорируется, однако зависимые Entity могут поменять статус на managed, если у них есть аннотации каскадных изменений;
- detached → exception сразу или на этапе commit-а транзакции;
- removed → managed, но только в рамках одной транзакции.

Как влияет операция remove на Entity-объекты каждого из четырех статусов?

- new → операция игнорируется, однако зависимые Entity могут поменять статус на removed, если у них есть аннотации каскадных изменений и они имели статус managed;
- managed → removed, запись объекта в базе данных будет удалена при commit-е транзакции (также произойдут операции remove для всех каскаднозависимых объектов);
- detached → exception сразу или на этапе commit-а транзакции;
- removed → операция игнорируется.

Как влияет операция merge на Entity-объекты каждого из четырех статусов?

- new → будет создан новый managed entity, в который будут скопированы данные прошлого объекта;
- managed → операция игнорируется, однако операция merge сработает на каскаднозависимые Entity, если их статус не managed;
- detached → либо данные будут скопированы в существующий managed entity с тем же первичным ключом, либо создан новый managed, в который скопируются данные;
- removed → exception сразу или на этапе commit-а транзакции.

Как влияет операция refresh на Entity-объекты каждого из четырех статусов?

- managed → будут восстановлены все изменения из базы данных данного Entity, также произойдет refresh всех каскаднозависимых объектов;
- new, removed, detached → exception.

Как влияет операция detach на Entity-объекты каждого из четырех статусов?

- managed, removed → detached;
- new, detached → операция игнорируется.

Для чего нужна аннотация Basic?

@Basic указывает на простейший тип маппинга данных на колонку таблицы базы данных. В параметрах аннотации можно указать **fetch** стратегию доступа к полю и является ли это поле обязательным или нет. Может быть применена к полю любого из следующих типов:

- примитивы и их обертки;
- java.lang.String;
- java.math.BigInteger;
- java.math.BigDecimal;
- java.util.Date;
- java.util.Calendar;
- java.sql.Date;
- java.sql.Time;
- java.sql.Timestamp;
- byte[] or Byte[];
- char[] or Character[];
- enums;
- любые другие типы, которые реализуют Serializable.

Аннотацию @Basic можно не ставить, как это и происходит по умолчанию.

Аннотация @Basic определяет 2 атрибута:

1. **optional** – boolean (по умолчанию true) – определяет, может ли значение поля или свойства быть null. Игнорируется для примитивных типов. Но если тип поля не примитивного типа, то при попытке сохранения сущности будет выброшено исключение.

2. **fetch** – FetchType (по умолчанию EAGER) – определяет, должен ли этот атрибут извлекаться незамедлительно (EAGER) или лениво (LAZY). Это необязательное требование JPA, и провайдерам разрешено незамедлительно загружать данные, даже для которых установлена ленивая загрузка.

Без аннотации @Basic при получении сущности из БД по умолчанию ее поля базового типа загружаются принудительно (EAGER) и значения этих полей могут быть null.

Для чего нужна аннотация Column?

@Column сопоставляет поле класса столбцу таблицы, а ее атрибуты определяют поведение в этом столбце, используется для генерации схемы базы данных.

@Basic vs @Column:

1. Атрибуты @Basic применяются к сущностям JPA, тогда как атрибуты @Column применяются к столбцам базы данных.

2. @Basic имеет атрибут **optional**, который говорит о том, может ли поле объекта быть **null** или нет; с другой стороны атрибут **nullable** аннотации @Column указывает, может ли соответствующий столбец в таблице быть null.

3. Можно использовать @Basic, чтобы указать, что поле должно быть загружено лениво.

4. Аннотация @Column позволяет указать имя столбца в таблице и ряд других свойств:

- **insertable/updatable** – можно ли добавлять/изменять данные в колонке, по умолчанию true;
- **length** – длина, для строковых типов данных, по умолчанию 255.

Коротко, в @Column задаем **constraints**, а в @Basic – **FetchTypes**.

Для чего нужна аннотация Access?

Определяет тип доступа к полям сущности. Для чтения и записи этих полей есть два подхода:

1. **Field access (доступ по полям).** При таком способе аннотации маппинга (Id, Column,...) размещаются над полями, и Hibernate напрямую работает с полями сущности, читая и записывая их.

2. **Property access (доступ по свойствам)**. При таком способе аннотации размещаются над методами-геттерами, но не над сеттерами.

По умолчанию тип доступа определяется местом, в котором находится аннотация @Id. Если она будет над полем – это будет AccessType.FIELD, если над геттером – это AccessType.PROPERTY.

Чтобы явно определить тип доступа у сущности, нужно использовать аннотацию @Access, которая может быть указана у сущности, Mapped Superclass и Embeddable class, а также над полями или методами.

Поля, унаследованные от суперкласса, имеют тип доступа этого суперкласса.

Если у одной сущности определены разные типы доступа, то нужно использовать аннотацию @Transient для избежания дублирования маппинга.

Для чего нужна аннотация @Cacheable?

@Cacheable – необязательная аннотация JPA, используется для указания того, должна ли сущность храниться в кеше второго уровня.

В JPA говорится о пяти значениях shared-cache-mode из persistence.xml, который определяет как будет использоваться second-level cache:

- **ENABLE_SELECTIVE**: только сущности с аннотацией @Cacheable (равносильно значению по умолчанию @Cacheable(value = true)) будут сохраняться в кеше второго уровня;
- **DISABLE_SELECTIVE**: все сущности будут сохраняться в кеше второго уровня, за исключением сущностей, помеченных @Cacheable(value = false) как некешируемые;
- **ALL**: сущности всегда кешируются, даже если они помечены как некешируемые;
- **NONE**: ни одна сущность не кешируется, даже если помечена как кешируемая. При данной опции имеет смысл вообще отключить кеш второго уровня;
- **UNSPECIFIED**: применяются значения по умолчанию для кеша второго уровня, определенные Hibernate. Это эквивалентно тому, что вообще не используется shared-cache-mode, так как Hibernate не включает кеш второго уровня, если используется режим UNSPECIFIED.

Аннотация @Cacheable размещается над классом сущности. Ее действие распространяется на эту сущность и ее наследников, если они не определили другое поведение.

Для чего нужны аннотации @Embedded и @Embeddable?

@Embeddable – аннотация JPA, размещается над классом для указания того, что класс является встраиваемым в другие классы.

@Embedded – аннотация JPA, используется для размещения над полем в классе-сущности для указания того, что внедряется встраиваемый класс.

Как смапить составной ключ?

Составной первичный ключ, также называемый составным ключом, представляет собой комбинацию из двух или более столбцов для формирования первичного ключа таблицы.

@IdClass

Допустим, есть таблица с именем Account, и она имеет два столбца – accountNumber и accountType, которые формируют составной ключ. Чтобы обозначить оба этих поля как части составного ключа, необходимо создать класс, например, ComplexKey с этими полями.

Затем нужно аннотировать сущность Account аннотацией @IdClass(ComplexKey.class) и объявить поля из класса ComplexKey в сущности Account с такими же именами и аннотировать их с помощью @Id.

@EmbeddedId

Допустим, что необходимо сохранить некоторую информацию о книге с заголовком и языком в качестве полей первичного ключа. В этом случае класс первичного ключа, BookId, должен быть аннотирован @Embeddable.

Затем нужно встроить этот класс в сущность Book, используя @EmbeddedId.

Для чего нужна аннотация ID? Какие @GeneratedValue вы знаете?

Аннотация @Id определяет простой (не составной) первичный ключ, состоящий из одного поля. В соответствии с JPA, допустимые типы атрибутов для первичного ключа:

- примитивные типы и их обертки;
- строки;

- BigDecimal и BigInteger;
- java.util.Date и java.sql.Date.

Если хотим, чтобы значение первичного ключа генерировалось автоматически, необходимо добавить первичному ключу, отмеченному аннотацией @Id, аннотацию @GeneratedValue.

Возможны 4 варианта:

1. **AUTO (default).** Указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая ее диалект, так как у разных БД разные способы по умолчанию. Поведение по умолчанию – исходить из типа поля идентификатора.

2. **IDENTITY.** Для генерации значения первичного ключа будет использоваться столбец IDENTITY, имеющийся в базе данных. Значения в столбце автоматически увеличиваются вне текущей выполняемой транзакции(на стороне базы, так что этого столбца не увидим, что позволяет базе данных генерировать новое значение при каждой операции вставки. В промежутках транзакций сущность будет сохранена.

3. **SEQUENCE.** Тип генерации, рекомендуемый документацией Hibernate. Для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). В БД можно будет увидеть дополнительную таблицу. Но если БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE. В промежутках транзакций сущность не будет сохранена, так как Hibernate возьмет из таблицы id hibernate-sequence и вернется обратно в приложение. SEQUENCE – это объект базы данных, который генерирует инкрементные целые числа при каждом последующем запросе.

4. **TABLE.** Hibernate должен получать первичные ключи для сущностей из создаваемой для этих целей таблицы, способной содержать именованные сегменты значений для любого количества сущностей. Требует использования пессимистических блокировок, которые помещают все транзакции в последовательный порядок и замедляет работу приложения.

Расскажите про аннотации @JoinColumn и @JoinTable? Где и для чего они используются?

@JoinColumn используется для указания столбца FOREIGN KEY, используемого при установлении связей между сущностями или коллекциями. Только сущность-владелец связи может иметь внешние ключи от другой сущности (владеемой). Но можно указать @JoinColumn как во владеющей таблице, так и во владеемой, но столбец с внешними ключами все равно появится во владеющей таблице.

Особенности использования:

- **@OneToOne**: означает, что появится столбец в таблице сущности-владельца связи, который будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности;
- **@OneToMany/@ManyToOne**: если не указать на владеемой стороне связи @mappedBy, создается joinTable с ключами обеих таблиц. Но при этом же у владельца создается столбец с внешними ключами.

@JoinColumns используется для группировки нескольких аннотаций @JoinColumn, которые используются при установлении связей между сущностями или коллекциями, у которых составной первичный ключ и требуется несколько колонок для указания внешнего ключа.

В каждой аннотации @JoinColumn должны быть указаны элементы name и referencedColumnName.

@JoinTable используется для указания связывающей (сводной, третьей) таблицы между двумя другими таблицами.

Для чего нужны аннотации @OrderBy и @OrderColumn, чем они отличаются?

@OrderBy указывает порядок, в соответствии с которым должны располагаться элементы коллекций сущностей, базовых или встраиваемых типов при их извлечении из БД. Если в кеше есть нужные данные, то сортировки не будет, так как @OrderBy просто добавляет к sql-запросу Order By, а при получении данных из кеша, обращения к БД нет. Эта аннотация может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany.

При использовании с коллекциями базовых типов, которые имеют аннотацию @ElementCollection, элементы этой коллекции будут отсортированы в натуральном порядке, по значению базовых типов.

Если это коллекция встраиваемых типов (@Embeddable), то, используя точку ("."), можно сослаться на атрибут внутри встроенного атрибута.

Если это коллекция сущностей, то у аннотации @OrderBy можно указать имя поля сущности, по которому сортировать эти сущности:

Если не указывать у @OrderBy параметр, то сущности будут упорядочены по первичному ключу.

В случае с сущностями доступ к полю по точке (".") не работает. Попытка использовать вложенное свойство, например, @OrderBy ("supervisor.name") повлечет Runtime Exceprtion.

@OrderColumn создает в таблице столбец с индексами порядка элементов, который используется для поддержания постоянного порядка в списке, но этот столбец не считается частью состояния сущности или встраиваемого класса.

Hibernate отвечает за поддержание порядка как в базе данных при помощи столбца, так и при получении сущностей и элементов из БД. Hibernate отвечает за обновление порядка при записи в базу данных, чтобы отразить любое добавление, удаление или иное изменение порядка, влияющее на список в таблице.

@OrderBy vs @OrderColumn

Порядок, указанный в @OrderBy, применяется только в рантайме при выполнении запроса к БД, То есть в контексте персистентности, в то время как при использовании @OrderColumn, порядок сохраняется в отдельном столбце таблицы и поддерживается при каждой вставке/обновлении/удалении элементов.

Для чего нужна аннотация Transient?

@Transient используется для объявления того, какие поля у сущности, встраиваемого класса или Mapped SuperClass не будут сохранены в базе данных.

Persistent fields (постоянные поля) – это поля, значения которых будут по умолчанию сохранены в БД. Ими являются любые не static и не final поля.

Transient fields (временные поля):

- static и final поля сущностей;

- иные поля, объявленные явно с использованием Java-модификатора transient либо JPA-аннотации @Transient.

Какие шесть видов блокировок (lock) описаны в спецификации JPA (или какие есть значения у enum LockModeType в JPA)?

В порядке от самого ненадежного и быстрого, до самого надежного и медленного:

1. **NONE** – без блокировки.
2. **OPTIMISTIC** (синоним READ в JPA 1) – оптимистическая блокировка: если при завершении транзакции кто-то извне изменит поле @Version, то будет сделан RollBack транзакции и будет выброшено OptimisticLockException.
3. **OPTIMISTIC_FORCE_INCREMENT** (синоним WRITE в JPA 1) – работает по тому же алгоритму, что и LockModeType.OPTIMISTIC за тем исключением, что после commit значение поля Version принудительно увеличивается на 1. В итоге после каждого коммита поле увеличится на 2 (увеличение, которое можно увидеть в Post-Update + принудительное увеличение).
4. **PESSIMISTIC_READ** – данные блокируются в момент чтения, и это гарантирует, что никто в ходе выполнения транзакции не сможет их изменить. Остальные транзакции смогут параллельно читать эти данные. Использование этой блокировки может вызывать долгое ожидание блокировки или даже выкидывание PessimisticLockException.
5. **PESSIMISTIC_WRITE** – данные блокируются в момент записи, и никто с момента захвата блокировки не может в них писать и не может их читать до окончания транзакции, владеющей блокировкой. Использование этой блокировки может вызывать долгое ожидание блокировки.
6. **PESSIMISTIC_FORCE_INCREMENT** – ведет себя как PESSIMISTIC_WRITE, но в конце транзакции увеличивает значение поля @Version, даже если фактически сущность не изменилась.

Оптимистичное блокирование – подход предполагает, что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным, позволяет им свободно выполнять любые чтения и обновления данных. Но при окончании транзакции производится проверка, изменились ли данные в ходе выполнения данной транзакции и, если да, транзакция обрывается и выбрасывается OptimisticLockException. Оптимистичное блокирование в JPA реализовано с помощью внедрения в сущность специального поля версии:

@Version

private long version;

Поле, аннотирование @Version, может быть целочисленным или временнЫм. При завершении транзакции, если сущность была заблокирована оптимистично, будет проверено, не изменилось ли значение @Version кем-либо еще после того, как данные были прочитаны, и, если изменилось, будет выкинуто OptimisticLockException. Использование этого поля позволяет отказаться от блокировок на уровне базы данных и сделать все на уровне JPA, улучшая уровень конкурентности.

Позволяет отказаться от блокировок на уровне БД и делать все с JPA.

Пессимистичное блокирование – подход ориентирован на транзакции, которые часто конкурируют за одни и те же данные, поэтому блокируется доступ к данным в тот момент,

когда происходит чтение. Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным, и ждут снятия блокировки (или кидают исключение). Пессимистичное блокирование выполняется на уровне базы и поэтому не требует вмешательств в код сущности.

Блокировки ставятся с помощью вызова метода lock() у EntityManager, в который передается сущность, требующая блокировки и уровень блокировки:

EntityManager em = entityManagerFactory.createEntityManager();

em.lock(company1, LockModeType.OPTIMISTIC);

Какие два вида кэшей (cache) вы знаете в JPA и для чего они нужны?

- **first-level cache** (кеш первого уровня) кеширует данные одной транзакции;
- **second-level cache** (кеш второго уровня) кэширует данные транзакций от одной фабрики сессий. Провайдер JPA может, но не обязан реализовывать работу с кешем второго уровня.

Кеш первого уровня – это кеш сессии (Session), который является обязательным, это и есть PersistenceContext. Через него проходят все запросы.

Если выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление этого объекта для того, чтобы сократить количество выполненных запросов в БД. Например, при пяти обращении к одному и тому же объекту из БД в рамках одного persistence context, запрос в БД будет выполнен один раз, а остальные четыре загрузки будут выполнены из кеша. Если закроем сессию, то все объекты, находящиеся в кеше, теряются, а далее – либо сохраняются в БД, либо обновляются.

Особенности кеша первого уровня:

- включен по умолчанию, его нельзя отключить;
- связан с сессией (контекстом персистентности), то есть разные сессии видят только объекты из своего кеша и не видят объекты, находящиеся в кешах других сессий;
- при закрытии сессии PersistenceContext очищается – кешированные объекты, находившиеся в нем, удаляются;
- при первом запросе сущности из БД она загружается в кеш, связанный с этой сессией;
- если в рамках этой же сессии снова запросим эту же сущность из БД, то она будет загружена из кеша и повторного SQL-запроса в БД сделано не будет;
- сущность можно удалить из кеша сессии методом evict(), после чего следующая попытка получить эту же сущность повлечет обращение к базе данных;
- метод clear() очищает весь кеш сессии.

Если кеш первого уровня привязан к объекту сессии, то **кеш второго уровня** привязан к объекту-фабрике сессий (Session Factory object), поэтому кеш второго уровня доступен одновременно в нескольких сессиях или контекстах персистентности. Кеш второго уровня требует некоторой настройки и поэтому не включен по умолчанию. Настройка кеша заключается в конфигурировании реализации кеша и разрешения сущностям быть закешированными.

Hibernate не реализует сам никакого in-memory cache, а использует существующие реализации кешей.

Как работать с кешем 2 уровня?

Чтение из кеша второго уровня происходит только в том случае, если нужный объект не был найден в кеше первого уровня.

Hibernate поставляется со встроенной поддержкой стандарта кеширования Java JCache, а также двух популярных библиотек кеширования: **Ehcache** и **Infinispan**.

В Hibernate кеширование второго уровня реализовано в виде абстракции, то есть необходимо предоставить любую ее реализацию. Например, можно использовать следующих провайдеров: Ehcache, OSCache, SwarmCache, JBoss TreeCache. Для Hibernate требуется только реализация интерфейса org.hibernate.cache.spi.RegionFactory, который инкапсулирует все детали, относящиеся к конкретным провайдерам. По сути RegionFactory действует как мост между Hibernate и поставщиками кеша. В качестве примера воспользуемся Ehcache. Для этого:

- добавим мавен-зависимость кеш-провайдера нужной версии;
- включим кеш второго уровня и определить конкретного провайдера;

hibernate.cache.use_second_level_cache=true

hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

- установим у нужных сущностей JPA-аннотацию @Cacheable, обозначающую, что сущность нужно кешировать, и Hibernate-аннотацию @Cache, настраивающую детали кеширования, у которой в качестве параметра указать стратегию параллельного доступа.

Стратегии параллельного доступа к объектам:

Проблема заключается в том, что кеш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут одновременно в разных транзакциях работать с одним и тем же объектом. Следовательно надо как-то обеспечивать их одинаковым представлением этого объекта.

- **READ_ONLY**: Используется только для сущностей, которые никогда не изменяются (будет выброшено исключение, если попытаться обновить такую сущность). Просто и производительно. Подходит для некоторых статических данных, которые не меняются.
- **NONSTRICT_READ_WRITE**: Кеш обновляется после совершения транзакции, которая изменила данные в БД и закоммитила их. Таким образом, строгая согласованность не гарантируется, и существует небольшое временное окно между обновлением данных в БД и обновлением тех же данных в кеше, во время которого параллельная транзакция может получить из кеша устаревшие данные.
- **READ_WRITE**: Гарантирует строгую согласованность, которая достигается за счет «мягких» блокировок: когда обновляется кешированная сущность, на нее накладывается мягкая блокировка, которая снимается после коммита транзакции. Все параллельные транзакции, которые пытаются получить доступ к записям в кеше с наложенной мягкой блокировкой, не смогут их прочитать или записать и отправят запрос в БД. Ehcache использует эту стратегию по умолчанию.

- **TRANSACTIONAL**: полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, словно они работали с ними последовательно одна транзакция за другой. Плата за это – блокировки и потеря производительности.

Что такое JPQL/HQL и чем он отличается от SQL?

Hibernate Query Language (HQL) и **Java Persistence Query Language (JPQL)** являются объектно-ориентированными языками запросов, схожими по природе с SQL. JPQL – это подмножество HQL.

JPQL – это язык запросов, практически такой же, как SQL, но вместо имен и колонок таблиц базы данных использует имена классов Entity и их атрибуты. В качестве параметров запросов используются типы данных атрибутов Entity, а не полей баз данных. В отличие от SQL в JPQL есть автоматический полиморфизм, то есть каждый запрос к Entity возвращает не только объекты этого Entity, но и объекты всех его классов-потомков, независимо от стратегии наследования. В JPA запрос представлен в виде javax.persistence.Query или javax.persistence.TypedQuery, полученных из EntityManager.

В Hibernate HQL-запрос представлен org.hibernate.query.Query, полученный из Session. Если HQL является именованным запросом, то будет использоваться Session#getNamedQuery, в противном случае требуется Session#createQuery.

Что такое Criteria API и для чего он используется?

Начиная с версии 5.2, Hibernate Criteria API объявлен deprecated. Вместо него рекомендуется использовать JPA Criteria API.

JPA Criteria API – это актуальный API, используемый только для выборки (select) сущностей из БД в более объектно-ориентированном стиле.

Основные преимущества JPA Criteria API:

- ошибки могут быть обнаружены во время компиляции;
- позволяет динамически формировать запросы на этапе выполнения приложения.

Основные недостатки:

- нет контроля над запросом, сложно отловить ошибку;
- влияет на производительность, множество классов.

Для динамических запросов фрагменты кода создаются во время выполнения, поэтому JPA Criteria API является предпочтительней.

Некоторые области применения Criteria API:

- поддерживает проекцию, которую можно использовать для агрегатных функций вроде sum(), min(), max() и т. д.;
- может использовать ProjectionList для извлечения данных только из выбранных колонок;
- может быть использована для join запросов с помощью соединения нескольких таблиц, используя методы createAlias(), setFetchMode() и setProjection();
- поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод add(), с помощью которого добавляются ограничения (Restrictions).

- позволяет добавлять порядок (сортировку) к результату с помощью метода addOrder().

Расскажите про проблему N+1 Select и путях ее решения

Проблема N+1 запросов возникает, когда получение данных из БД выполняется за N дополнительных SQL-запросов для извлечения тех же данных, которые могли быть получены при выполнении основного SQL-запроса.

1. JOIN FETCH

И при FetchType.EAGER, и при FetchType.LAZY поможет JPQL-запрос с JOIN FETCH. Опцию «FETCH» можно использовать в JOIN (INNER JOIN или LEFT JOIN) для выборки связанных объектов в одном запросе вместо дополнительных запросов для каждого доступа к ленивым полям объекта.

Лучший вариант решения для простых запросов (1-3 уровня вложенности связанных объектов).

```
select pc
   from PostComment pc
   join fetch pc.post p
```

2. EntityGraph

Если нужно получить много данных через jpql-запрос, лучше всего использовать EntityGraph.

3. @Fetch(FetchMode.SUBSELECT)

Аннотация Hibernate. Можно использовать только с коллекциями. Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций:

```
@Fetch(value = FetchMode.SUBSELECT)
   private Set<Order> orders = new HashSet<>();
```

4. Batch fetching

Аннотация Hibernate, в JPA ее нет. Указывается над классом сущности или над полем коллекции с ленивой загрузкой. Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций. Количество загружаемых сущностей указывается в аннотации.

```
@BatchSize(size=5)
private Set<Order> orders = new HashSet<>();
```

5. HibernateSpecificMapping, SqlResultSetMapping

Рекомендуется использовать для нативных запросов.

Что такое EntityGraph? Как и для чего их использовать?

Основная цель JPA Entity Graph – улучшить производительность в рантайме при загрузке базовых полей сущности и связанных сущностей и коллекций.

Hibernate загружает весь граф в одном SELECT-запросе, то есть все указанные связи от нужной сущности. Если необходимо загрузить дополнительные сущности, находящиеся в связанных сущностях, используется **Subgraph**.

EntityGraph можно определить с помощью аннотации **@NamedEntityGraph** для Entity, она определяет уникальное имя и список атрибутов (attributeNodes), которые должны быть загружены с использованеим entityManager из JPA API:

EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);

entityGraph.addAttributeNodes("subject");

entityGraph.addAttributeNodes("user");

entityGraph.addSubgraph("comments").addAttributeNodes("user");

JPA определяет два свойства или подсказки, с помощью которых Hibernate может выбирать стратегию извлечения графа сущностей во время выполнения:

- **fetchgraph** – все атрибуты, перечисленные в EntityGraph, меняют fetchType на EAGER, все остальные – на LAZY;
- **loadgraph** – все атрибуты, перечисленные в EntityGraph, меняют fetchType на EAGER, все остальные сохраняют свой fetchType. С помощью NamedSubgraph можно изменить fetchType вложенных объектов Entity.

Загрузить EntityGraph можно тремя способами:

1. Используя перегруженный метод find(), который принимает Map с настройками EntityGraph.
2. Используя JPQL и передав подсказку через setHint().
3. С помощью Criteria API.

Мемоизация

Memoization – вариант кеширования, заключающийся в том, что для функции создается таблица результатов. Результат функции, вычисленной при определенных значениях параметров, заносится в эту таблицу. В дальнейшем результат берется из данной таблицы.

Эта техника позволяет за счет использования дополнительной памяти ускорить работу программы.

Можно применить только к функциям, которые являются:

- детерминированными (т. е. при одном и том же наборе параметров функции должны возвращать одинаковое значение);
- без побочных эффектов (т. е. не должны влиять на состояние системы).

В Java наиболее подходящей кандидатурой на роль хранилища является интерфейс Map. Сложность операций get,put,contains равна O(1). Это позволяет гарантировать ограничение задержки при выполнении мемоизации.

Мемоизация реализована в библиотеке ehcache.

Spring

Что такое инверсия контроля (IoC) и внедрение зависимостей (DI)? Как эти принципы реализованы в Spring?

Inversion of Control – подход, который позволяет конфигурировать и управлять объектами Java с помощью рефлексии. Вместо ручного внедрения зависимостей фреймворк забирает ответственность за это через IoC-контейнер. Контейнер отвечает за управление жизненным циклом объектов: создание объектов, вызов методов инициализации и конфигурирование объектов через связывание их между собой.

Объекты, создаваемые контейнером, называются beans. Конфигурирование контейнера осуществляется через внедрение аннотаций, но есть возможность, по старинке, загрузить XML-файлы, содержащие определение bean'ов и предоставляющие информацию, необходимую для создания bean'ов.

Dependency Injection является одним из способов реализации принципа IoC в Spring. Это шаблон проектирования, в котором контейнер передает экземпляры объектов по их типу другим объектам с помощью конструктора или метода класса (setter), что позволяет писать слабосвязный код.

Что такое IoC контейнер?

В среде Spring IoC-контейнер представлен интерфейсом **ApplicationContext**, который является оберткой над **BeanFactory**, предоставляющей дополнительные возможности, например AOP и транзакции. Интерфейс BeanFactory предоставляет фабрику для бинов, которая в то же время является IoC-контейнером приложения. Управление бинами основано на конфигурации (аннотации или xml). Контейнер создает объекты на основе конфигураций и управляет их жизненным циклом от создания объекта до уничтожения.

Расскажите про ApplicationContext и BeanFactory, чем отличаются? В каких случаях что стоит использовать?

Функционал	BeanFactory	ApplicationContext
Инициализация/автоматическое связывание бина	Да	Да
Автоматическая регистрация `BeanPostProcessor`	Нет	Да
Автоматическая регистрация `BeanFactoryPostProcessor`	Нет	Да
Удобный доступ к `MessageSource` (для i18n)	Нет	Да
`ApplicationEvent` публикация	Нет	Да

ApplicationContext является наследником BeanFactory и полностью реализует его функционал, добавляя больше специфических enterprise-функций. Может работать с бинами всех скоупов.

BeanFactory – это фактический контейнер, который создает, настраивает и управляет рядом bean-компонентов. Эти бины обычно взаимодействуют друг с другом и имеют зависимости между собой. Эти зависимости отражены в данных конфигурации, используемых BeanFactory. Может работать с бинами **singleton** и **prototype**.

BeanFactory обычно используется тогда, когда ресурсы ограничены (мобильные устройства), так как он легче по сравнению с ApplicationContext. Поэтому, если ресурсы не сильно ограничены, то лучше использовать ApplicationContext.

ApplicationContext загружает все бины при запуске, а BeanFactory по требованию.

Расскажите про аннотацию @Bean?

Аннотация **@Bean** используется для указания того, что метод создает, настраивает и инициализирует новый объект, управляемый IoC-контейнером. Такие методы можно использовать как в классах с аннотацией **@Configuration**, так и в классах с аннотацией **@Component** (или ее наследниках).

Имеет следующие свойства:

- **destroyMethod, initMethod** – варианты переопределения методов инициализации и удаления бина при указании их имен в аннотации;
- **name** – имя бина, по умолчанию именем бина является имя метода;
- **value** – алиас для name().

Расскажите про аннотацию @Component?

@Component используется для указания класса в качестве компонента Spring. Такой класс будет сконфигурирован как spring Bean.

Чем отличаются аннотации @Bean и @Component?

@Bean ставится над методом и позволяет добавить bean, уже реализованного сторонней библиотекой класса, в контейнер, а **@Component** используется для указания класса, написанного программистом.

Расскажите про аннотации @Service и @Repository. Чем они отличаются?

@Repository указывает, что класс используется для работы с поиском, получением и хранением данных. Аннотация может использоваться для реализации шаблона DAO.

@Service указывает, что класс является сервисом для реализации бизнес-логики.

@Repository, @Service, @Controller и @Configuration являются алиасами @Component, их также называют стереотипными аннотациями.

Задача @Repository заключается в том, чтобы отлавливать определенные исключения персистентности и пробрасывать их как одно непроверенное исключение Spring Framework. Для этого в контекст должен быть добавлен класс **PersistenceExceptionTranslationPostProcessor**.

Расскажите про аннотацию @Autowired

@Autowired – автоматическое внедрение подходящего бина:

1. Контейнер определяет тип объекта для внедрения.
2. Контейнер ищет соответствующий тип бина в контексте (он же контейнер).
3. Если есть несколько кандидатов и один из них помечен как @Primary, то внедряется он.

4. Если используется @Qualifier, то контейнер будет использовать информацию из @Qualifier, чтобы понять, какой компонент внедрять.
5. В противном случае контейнер внедрит бин, основываясь на его имени или ID.
6. Если ни один из способов не сработал, то будет выброшено исключение.

Контейнер обрабатывает DI с помощью AutowiredAnnotationBeanPostProcessor. В связи с этим аннотация не может быть использована ни в одном BeanFactoryPP или BeanPP.

В аннотации есть один параметр required = true/fals. Он указывает, обязательно ли делать DI. По умолчанию true. Либо можно не выбрасывать исключение, а оставить поле с null, если нужный бин не был найден – false.

При циклической зависимости, когда объекты ссылаются друг на друга, нельзя ставить над конструктором.

Однако при внедрении прямо в поля не нужно предоставлять прямого способа создания экземпляра класса со всеми необходимыми зависимостями. Это означает, что:

- существует способ (через вызов конструктора по умолчанию) создать объект с использованием **new** в состоянии, когда ему не хватает некоторых из его обязательных зависимостей, и использование приведет к NullPointerException;
- такой класс не может быть использован вне DI-контейнеров (тесты, другие модули) и нет способа кроме рефлексии предоставить ему необходимые зависимости;
- неизменность;

В отличие от способа с использованием конструктора внедрение через поля не может использоваться для присвоения зависимостей final-полям, что приводит к тому, что объекты становятся изменяемыми.

Расскажите про аннотацию @Resource

@Resource (аннотация java) пытается получить зависимость: по имени, по типу, затем по описанию (Qualifier). Имя извлекается из имени аннотируемого сеттера или поля либо берется из параметра name.

```
@Resource //По умолчанию поиск бина с именем "context"
private ApplicationContext context;


@Resource(name="greetingService") //Поиск бина с именем "greetingService"
public void setGreetingService(GreetingService service) {
    this.greetingService = service;
}
```

Отличие от @Autowired:

- ищет бин сначала по имени, а потом по типу;
- не нужна дополнительная аннотация для указания имени конкретного бина;

- @Autowired позволяет отметить место вставки бина как необязательное @Autowired(required = false);
- при замене Spring Framework на другой фреймворк менять аннотацию @Resource не нужно.

Расскажите про аннотацию @Inject

@Inject входит в пакет javax.inject. Чтобы ее использовать, нужно добавить зависимость:

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

@Inject (аннотация java) – аналог @Autowired (аннотация spring) в первую очередь пытается подключить зависимость по типу, затем по описанию и только потом по имени. В ней нет параметров. Поэтому при использовании конкретного имени (Id) бина используется @Named:

@Inject

@Named("yetAnotherFieldInjectDependency")

private ArbitraryDependency yetAnotherFieldInjectDependency;

Расскажите про аннотацию @Lookup

Обычно бины в приложении Spring являтся синглтонами и для внедрения зависимостей используется конструктор или сеттер.

Но бывает и другая ситуация: имеется бин Car – синглтон (singleton bean) – и ему требуется каждый раз новый экземпляр бина Passenger. То есть Car – синглтон, а Passenger – так называемый прототипный бин (prototype bean). Жизненные циклы бинов разные. Бин Car создается контейнером только раз, а бин Passenger создается каждый раз новый. Допустим, это происходит каждый раз при вызове какого-то метода бина Car. Вот здесь и пригодится внедрение бина с помощью метода Lookup. Оно происходит не при инициализации контейнера, а позднее: каждый раз, когда вызывается метод. Суть в том, что создается метод-заглушка в бине Car и он помечается специальным образом – аннотацией @Lookup. Этот метод должен возвращать бин Passenger, каждый раз новый. Контейнер Spring под капотом создаст подкласс и переопределит этот метод и будет выдавать новый экземпляр бина Passenger при каждом вызове аннотированного метода. Даже если в заглушке он возвращает null (а так и надо делать, все равно этот метод будет переопределен).

Можно ли вставить бин в статическое поле? Почему?

Spring не позволяет внедрять бины напрямую в статические поля. Это связано с тем, что когда загрузчик классов загружает статические значения, контекст Spring еще не загружен. Чтобы исправить это, можно создать нестатический сеттер-метод с @Autowired:

private static OrderItemService orderItemService;

@Autowired

```
  public void setOrderItemService(OrderItemService orderItemService) {
     TestDataInit.orderItemService = orderItemService;
  }
```

Расскажите про аннотации @Primary и @Qualifier

@Qualifier применяется, если кандидатов для автоматического связывания несколько, аннотация позволяет указать в качестве аргумента имя конкретного бина, который следует внедрить. Она может быть применена к отдельному полю класса, к отдельному аргументу метода или конструктора:

```
public class AutowiredClass {

   @Autowired //к полям класса
   @Qualifier("main")
   private GreetingService greetingService;

   @Autowired //к отдельному аргументу конструктора или метода
   public void prepare(@Qualifier("main") GreetingService greetingService){
      /* что-то делаем... */
   };
}
```

Поэтому у одной из реализации **GreetingService** должна быть установлена соответствующая аннотация @Qualifier:

```
@Component
@Qualifier("main")
public class GreetingServiceImpl implements GreetingService {
   //...
}
```

@Primary тоже используется, чтобы отдавать предпочтение бину, когда есть несколько бинов одного типа, но в ней нельзя задать имя бина, она определяет значение по умолчанию, в то время как @Qualifier более специфичен.

Если присутствуют аннотации @Qualifier и @Primary, то аннотация @Qualifier будет иметь приоритет.

Как заинжектить примитив?

Для этого можно использовать аннотацию **@Value**. Можно ставить над полем, конструктором, методом.

Такие значения можно получать из property файлов, из бинов, и т. п.

@Value("${some.key}")

public String stringWithDefaultValue;

В эту переменную будет внедрена строка, например, из property или из view.

Кроме того, для внедрения значений можно использовать язык SpEL (Spring Expression Language).

Как заинжектить коллекцию?

Если внедряемый объект массив, коллекция или map с дженериком, то, используя аннотацию @Autowired, Spring внедрит все бины, подходящие по типу в этот массив (или другую структуру данных). В случае с map ключом будет имя бина.

Используя аннотацию @Qualifier можно настроить тип искомого бина.

Бины могут быть упорядочены, если вставляются в списки (не Set или Map) или массивы. Поддерживаются как аннотация @Order, так и интерфейс Ordered.

Расскажите про аннотацию @Conditional

Spring предоставляет возможность на основе используемого алгоритма включить или выключить определение бина или всей конфигурации через @Conditional, в качестве параметра которой указывается класс, реализующий интерфейс Condition, с единственным методом matches(ConditionContext var1, AnnotatedTypeMetadata var2), возвращающий boolean.

Для создания более сложных условий можно использовать классы AnyNestedCondition, AllNestedConditions и NoneNestedConditions.

Аннотация @Conditional указывает, что компонент имеет право на регистрацию в контексте только тогда, когда все условия соответствуют.

Условия проверяются непосредственно перед тем, как должен быть зарегистрирован BeanDefinition компонента, и они могут помешать регистрации данного BeanDefinition. Поэтому при проверке условий нельзя допускать взаимодействия с бинами, которых еще не существует, с их BeanDefinition-ами можно.

Для того, чтобы проверить несколько условий, можно передать в @Conditional несколько классов с условиями:

@Conditional(HibernateCondition.class, OurConditionClass.class)

Если класс @Configuration помечен как @Conditional, то на все методы @Bean, аннотации @Import и аннотации @ComponentScan, связанные с этим классом, также будут распространяться указанные условия.

Расскажите про аннотацию @Profile

Профили – это ключевая особенность Spring Framework, позволяющая относить бины к разным профилям (логическим группам), например, dev, local, test, prod.

Можно активировать разные профили в разных средах, чтобы загрузить только те бины, которые нужны.

Используя аннотацию @Profile относим бин к конкретному профилю. Ее можно применять на уровне класса или метода. Аннотация @Profile принимает в качестве аргумента имя одного или нескольких профилей. Она фактически реализована с помощью более гибкой аннотации @Conditional.

Ее можно ставить на @Configuration и Component классы.

Расскажите про жизненный цикл бина, аннотации @PostConstruct и @PreDestroy()

1. Парсирование конфигурации и создание BeanDefinition.

- xml-конфигурация – ClassPathXmlApplicationContext(“context.xml”);
- конфигурация через аннотации с указанием пакета для сканирования – AnnotationConfigApplicationContext(“package.name”);
- конфигурация через аннотации с указанием класса (или массива классов), помеченного аннотацией @Configuration – AnnotationConfigApplicationContext(JavaConfig.class), этот способ конфигурации называется JavaConfig;
- groovy-конфигурация – GenericGroovyApplicationContext(“context.groovy”).

Если заглянуть внутрь AnnotationConfigApplicationContext, то можно увидеть два поля.

private final AnnotatedBeanDefinitionReader rea*der;*

*private final ClassPathBeanDefinitionScanner sc*anner;

ClassPathBeanDefinitionScanner сканирует указанный пакет на наличие классов, помеченных аннотацией @Component (или ее алиаса). Найденные классы парсируются и для них создаются BeanDefinition. Чтобы было запущено сканирование, в конфигурации должен быть указан пакет для сканирования @ComponentScan({"package.name"}). AnnotatedBeanDefinitionReader работает в несколько этапов.

Первый этап – это регистрация всех @Configuration для дальнейшего парсирования. Если в конфигурации используются Conditional, то будут зарегистрированы только те конфигурации, для которых Condition вернет true.

Второй этап – это регистрация BeanDefinitionRegistryPostProcessor, который при помощи класса ConfigurationClassPostProcessor парсирует JavaConfig и создает BeanDefinition.

Цель первого этапа – это создание всех BeanDefinition. BeanDefinition – это специальный интерфейс, через который можно получить доступ к метаданным будущего бина. В зависимости от конфигурации будет использоваться тот или иной механизм парсирования конфигурации.

BeanDefinition – это объект, который хранит в себе информацию о бине. Сюда входит: из какого класса бин надо создать, scope, установлена ли ленивая инициализация, нужно ли перед данным бином инициализировать другой, init и destroy методы, зависимости. Все полученные BeanDefinition’ы складываются в ConcurrentHashMap, в которой ключом является имя бина, а объектом – сам BeanDefinition. При старте приложения в IoC контейнер попадут бины, которые имеют scope Singleton (устанавливается по- молчанию), остальные создаются тогда, когда они нужны.

2. Настройка созданных BeanDefinition.

Есть возможность повлиять на бины до их создания, т. е. получить доступ к метаданным класса. Для этого существует специальный интерфейс BeanFactoryPostProcessor, реализовав который получаем доступ к созданным BeanDefinition и можем их изменять. В нем один метод.

Метод **postProcessBeanFactory** принимает параметром ConfigurableListableBeanFactory. Данная фабрика содержит много полезных методов, в том числе getBeanDefinitionNames, через который можно получить все BeanDefinitionNames, а уже потом по конкретному имени получить BeanDefinition для дальнейшей обработки метаданных.

Разберем одну из родных реализаций интерфейса BeanFactoryPostProcessor. Обычно настройки подключения к базе данных выносятся в отдельный property-файл, потом при помощи PropertySourcesPlaceholderConfigurer они загружаются и делается inject этих значений в нужное поле. Так как inject делается по ключу, то до создания экземпляра бина нужно заменить этот ключ на само значение из property-файла. Эта замена происходит в классе, который реализует интерфейс BeanFactoryPostProcessor. Название этого класса – PropertySourcesPlaceholderConfigurer. Он должен быть объявлен как static:

```
@Bean
public static PropertySourcesPlaceholderConfigurer configurer() {
    return new PropertySourcesPlaceholderConfigurer();
}
```

3. Создание кастомных FactoryBean.

FactoryBean – это generic-интерфейс, которому можно делегировать процесс создания бинов определенного типа. Когда конфигурация была исключительно в xml, разработчикам был необходим механизм, с помощью которого они бы могли управлять процессом создания бинов. Именно для этого и был сделан этот интерфейс.

Создадим фабрику, которая будет отвечать за создание всех бинов типа Color:

```
public class ColorFactory implements FactoryBean<Color> {
    @Override
    public Color getObject() throws Exception {
        Random random = new Random();
        Color color = new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));
        return color;
    }

    @Override
    public Class<?> getObjectType() {
        return Color.class;
    }

    @Override
```

```
    public boolean isSingleton() {
        return false;
    }
}
```

Теперь создание бина типа Color.class будет делегироваться ColorFactory, у которого при каждом создании нового бина будет вызываться метод getObject.

Для тех, кто пользуется JavaConfig, этот интерфейс будет абсолютно бесполезен.

4. Создание экземпляров бинов.

Сначала BeanFactory из коллекции Map с объектами BeanDefinition достает те, из которых создает все BeanPostProcessor-ы (инфраструктурные бины), необходимые для настройки обычных бинов.

Создаются экземпляры бинов через BeanFactory на основе ранее созданных BeanDefinition.

Созданием экземпляров бинов занимается BeanFactory на основе ранее созданных BeanDefinition. Из Map<BeanName, BeanDefinition> получаем Map<BeanName, Bean>.

Создание бинов может делегироваться кастомным FactoryBean.

5. Настройка созданных бинов.

На данном этапе бины уже созданы, их можно лишь донастроить.

Интерфейс BeanPostProcessor позволяет вклиниться в процесс настройки бинов до того, как они попадут в контейнер. ApplicationContext автоматически обнаруживает любые бины с реализацией BeanPostProcessor и помечает их как «post-processors» для того, чтобы создать их определенным способом. Например, в Spring есть реализации BeanPostProcessor-ов, которые обрабатывают аннотации @Autowired, @Inject, @Value и @Resource.

Интерфейс несет в себе два метода: postProcessBeforeInitialization(Object bean, String beanName) и postProcessAfterInitialization(Object bean, String beanName). У обоих методов параметры абсолютно одинаковые. Разница только в порядке их вызова. Первый вызывается до init-метода, второй – после.

Как правило, BeanPostProcessor-ы, которые заполняют бины через маркерные интерфейсы или тому подобное, реализовывают метод postProcessBeforeInitialization(Object bean, String beanName), тогда как BeanPostProcessor-ы, которые оборачивают бины в прокси, обычно реализуют postProcessAfterInitialization(Object bean, String beanName).

Прокси – это класс-декорация над бином. Например, можно добавить логику бину, но джава-код уже скомпилирован, поэтому нужно на лету сгенерировать новый класс. Этим классом необходимо заменить оригинальный класс так, чтобы никто не заметил подмены.

Есть два варианта создания этого класса:

- либо он должен наследоваться от оригинального класса (CGLIB) и переопределять его методы, добавляя нужную логику;
- либо он должен имплементировать те же самые интерфейсы, что и первый класс (Dynamic Proxy).

По конвенции спринга, если какой-то из BeanPostProcessor-ов меняет что-то в классе, то он должен это делать на этапе postProcessAfterInitialization(). Таким образом есть уверенность,

что initMethod у данного бина работает на оригинальный метод до того, как на него накрутился прокси.

Хронология событий:

Сначала сработает метод postProcessBeforeInitialization() всех имеющихся BeanPostProcessor-ов.

Затем, при наличии, будет вызван метод, аннотированный @PostConstruct.

Если бин имплементирует InitializingBean, то Spring вызовет метод afterPropertiesSet(). Не рекомендуется к использованию как устаревший.

При наличии будет вызван метод, указанный в параметре initMethod аннотации @Bean.

В конце бины пройдут через postProcessAfterInitialization(Object bean, String beanName). Именно на данном этапе создаются прокси стандартными BeanPostProcessor-ами. Затем отработают кастомные BeanPostProcessor-ы и применят логику к прокси-объектам. После чего все бины окажутся в контейнере, который будет обязательно обновлен методом refresh().

Но даже после этого можно донастроить бины ApplicationListener-ами.

Теперь все.

6. Бины созданы.

Их можно получить с помощью метода ApplicationContext.getBean().

7. Закрытие контекста.

Когда контекст закрывается (метод close() из ApplicationContext), бин уничтожается. Если в бине есть метод, аннотированный @PreDestroy, то перед уничтожением вызовется этот метод.

Если в аннотации @Bean определен метод destroyMethod, то будет вызван и он.

Аннотация PostConstruct

Spring вызывает методы, аннотированные @PostConstruct, только один раз сразу после инициализации свойств компонента. За данную аннотацию отвечает один из BeanPostProcessoroв.

Метод, аннотированный @PostConstruct, может иметь любой уровень доступа, может иметь любой тип возвращаемого значения (хотя тип возвращаемого значения игнорируется Spring-ом), метод не должен принимать аргументы. Он также может быть статическим, но преимуществ такого использования метода нет, т. к. доступ у него будет только к статическим полям/методам бина, и в таком случае смысл его использования для настройки бина пропадает.

Одним из примеров использования @PostConstruct является заполнение базы данных. Например, во время разработки может потребоваться создание пользователей по умолчанию.

Аннотация PreDestroy

Метод, аннотированный @PreDestroy, запускается только один раз непосредственно перед тем, как Spring удаляет компонент из контекста приложения.

Как и в случае с @PostConstruct, методы, аннотированные @PreDestroy, могут иметь любой уровень доступа, но не могут быть статическими. Целью этого метода может быть освобождение ресурсов или выполнение любых других задач очистки до уничтожения бина, например, закрытие соединения с базой данных.

Класс, имплементирующий BeanPostProcessor, обязательно должен быть бином, поэтому его помечают аннотацией @Component.

Расскажите про скоупы бинов? Какой скоуп используется по умолчанию? Что изменилось в Spring 5?

SCOPE_SINGLETON – инициализация произойдет один раз на этапе поднятия контекста.

SCOPE_PROTOTYPE – инициализация будет выполняться каждый раз по запросу. Причем во втором случае бин будет проходить через все BeanPostProcessor-ы, что может значительно снизить производительность.

Существует 2 области видимости по умолчанию.

Singleton – область видимости по умолчанию. В контейнере будет создан только один бин, и все запросы на него будут возвращать один и тот же бин.

Prototype – приводит к созданию нового бина каждый раз, когда он запрашивается.

Для бинов со scope «prototype» Spring не вызывает метод destroy(), так как не берет на себя контроль полного жизненного цикла этого бина. Spring не хранит такие бины в своем контексте (контейнере), а отдает их клиенту и больше о них не заботится (в отличие от синглтон-бинов).

4 области видимости в веб-приложении.

Request – область видимости – 1 HTTP запрос. На каждый запрос создается новый бин.

Session – область видимости – 1 сессия. На каждую сессию создается новый бин.

Application – область видимости – жизненный цикл ServletContext.

WebSocket – область видимости – жизненный цикл WebSocket.

Жизненный цикл web csope полный.

В пятой версии Spring Framework не стало Global session scope. Но появились Application и WebSocket.

Расскажите про аннотацию @ComponentScan

Первый шаг для описания конфигурации Spring – это добавление аннотаций @Component или наследников.

Однако Spring должен знать, где искать их. В @ComponentScan указываются пакеты, которые должны сканироваться. Можно указать массив строк.

Spring будет искать бины и в их подпакетах.

Можно расширить это поведение с помощью параметров includeFilters и excludeFilters в аннотации.

Для ComponentScan.Filter доступно пять типов фильтров:

- ANNOTATION

- ASSIGNABLE_TYPE
- ASPECTJ
- REGEX
- CUSTOM

Можно, например, в каком-то ненужном классе в не нашей библиотеке создать для него фильтр, чтобы его бин не инициализировался.

Как спринг работает с транзакциями? Расскажите про аннотацию @Transactional

Хорошая статья – https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth

Коротко:

Spring создает прокси для всех классов, помеченных @Transactional (либо если любой из методов класса помечен этой аннотацией), что позволяет вводить транзакционную логику до и после вызываемого метода. При вызове такого метода происходит следующее:

- proxy, который создал Spring, создает persistence context (или соединение с базой);
- открывает в нем транзакцию и сохраняет в контексте нити исполнения (в ThreadLocal);
- по мере надобности все сохраненное достается и внедряется в бины.

Таким образом, если в коде есть несколько параллельных нитей, то будет и несколько параллельных транзакций, которые будут взаимодействовать друг с другом согласно уровням изоляции.

Значения атрибута **propagation** у аннотации:

REQUIRED – применяется по умолчанию. При входе в @Transactional метод будет использована уже существующая транзакция или создана новая транзакция, если никакой еще нет.

REQUIRES_NEW – новая транзакция всегда создается при входе метод, ранее созданные транзакции приостанавливаются до момента возврата из метода.

NESTED – корректно работает только с базами данных, которые умеют savepoints. При входе в метод в уже существующей транзакции создается savepoint, который по результатам выполнения метода будет либо сохранен, либо отменен. Все изменения, внесенные методом, подтвердятся только позднее с подтверждением всей транзакции. Если текущей транзакции не существует, будет создана новая.

MANDATORY – всегда используется существующая транзакция и кидается исключение, если текущей транзакции нет.

SUPPORTS – метод будет использовать текущую транзакцию, если она есть, либо будет исполнятся без транзакции, если ее нет.

NOT_SUPPORTED – при входе в метод текущая транзакция, если она есть, будет приостановлена, и метод будет выполняться без транзакции.

NEVER – явно запрещает исполнение в контексте транзакции. Если при входе в метод будет существовать транзакция, будет выброшено исключение

Остальные атрибуты:

rollbackFor = Exception.class – если какой-либо метод выбрасывает указанное исключение, контейнер всегда откатывает текущую транзакцию. По умолчанию отлавливает RuntimeException.

noRollbackFor = Exception.class – указание того, что любое исключение, кроме заданного, должно приводить к откату транзакции.

rollbackForClassName и noRollbackForClassName – для задания имен исключений в строковом виде.

readOnly – разрешает только операции чтения.

В свойстве transactionManager хранится ссылка на менеджер транзакций, определенный в конфигурации Spring.

timeOut – по умолчанию используется таймаут, установленный по умолчанию для базовой транзакционной системы. Сообщает менеджеру tx о продолжительности времени, чтобы дождаться простоя tx, прежде чем принять решение об откате не отвечающих транзакций.

isolation – уровень изолированности транзакций.

Подробно:

Для работы с транзакциями Spring Framework использует AOP-прокси:

Для включения возможности управления транзакциями нужно разместить аннотацию @EnableTransactionManagement у класса конфигурации @Configuration.

Она означает, что классы, помеченные @Transactional, должны быть обернуты аспектом транзакций. Отвечает за регистрацию необходимых компонентов Spring, таких как TransactionInterceptor и советы прокси. Регистрируемые компоненты помещают перехватчик в стек вызовов при вызове методов @Transactional. Если используем Spring Boot и имеем зависимости spring-data-* или spring-tx, то управление транзакциями будет включено по умолчанию.

Пропагейшн работает, только если метод вызывает другой метод в другом сервисе. Если метод вызывает другой метод в этом же сервисе, то используется this и вызов проходит мимо прокси. Это ограничение можно обойти при помощи self-injection.

Слой логики (Service) – лучшее место для @Transactional.

Если пометить @Transactional класс @Service, то все его методы станут транзакционными. Так, при вызове, например, метода save() произойдет примерно следующее:

1. Вначале имеем:

- класс TransactionInterceptor, у которого вызывается метод invoke(...), внутри которого вызывается метод класса-родителя TransactionAspectSupport: invokeWithinTransaction(...), в рамках которого происходит магия транзакций.
- TransactionManager: решает, создавать ли новый EntityManager и/или транзакцию.
- EntityManager proxy: EntityManager – это интерфейс, и то, что внедряется в бин в слое DAO на самом деле не является реализацией EntityManager. В это поле внедряется EntityManager proxy, который будет перехватывать обращение к полю EntityManager и делегировать выполнение конкретному EntityManager в рантайме. Обычно EntityManager proxy представлен классом SharedEntityManagerInvocationHandler.

2. Transaction Interceptor.

В TransactionInterceptor отработает код до работы метода save(), в котором будет определено, выполнить ли метод save() в пределах уже существующей транзакции БД или должна стартовать новая отдельная транзакция. TransactionInterceptor сам не содержит логики по принятию решения, решение начать новую транзакцию, если это нужно, делегируется TransactionManager. Грубо говоря, на данном этапе метод будет обернут в try-catch и будет добавлена логика до его вызова и после:

```
try {
  transaction.begin();       // логика до
  service.save();
  transaction.commit();      // логика после
} catch(Exception ex) {
  transaction.rollback();
  throw ex;
}
```

3. TransactionManager.

Менеджер транзакций должен предоставить ответ на два вопроса:

- должен ли создаться новый EntityManager?
- должна ли стартовать новая транзакция БД?

Решение принимается, основываясь на следующих фактах:

- выполняется ли хоть одна транзакция в текущий момент или нет;
- атрибута «propagation» в @Transactional.

Если TransactionManager решил создать новую транзакцию, тогда:

- создается новый EntityManager;
- EntityManager «привязывается» к текущему потоку (Thread);
- «получается» соединение из пула соединений БД;
- соединение «привязывается» к текущему потоку.

И EntityManager и соединение привязываются к текущему потоку, используя переменные ThreadLocal.

4. EntityManager proxy.

Если метод save() слоя Service делает вызов метода save() слоя DAO, внутри которого вызывается, например, entityManager.persist(), то не происходит вызов метода persist() напрямую у EntityManager, записанного в поле класса DAO. Вместо этого метод вызывает EntityManager proxy, который достает текущий EntityManager для потока, и у него вызывается метод persist().

5. Отрабатывает DAO-метод save().

6. TransactionInterceptor.

Отработает код после работы метода save(). Другими словами, будет принято решение по коммиту/откату транзакции.

Кроме того, если в рамках одного метода сервиса обращаемся не только к методу save(), а к разным методам Service и DAO, то все они буду работать в рамках одной транзакции, которая оборачивает данный метод сервиса.

Вся работа происходит через прокси-объекты разных классов. Представим, что у нас в классе сервиса только один метод с аннотацией @Transactional, а остальные нет. Если вызовем метод с @Transactional, из которого вызовем метод без @Transactional, то оба будут отработаны в рамках прокси и будут обернуты в нашу транзакционную логику. Однако, если вызовем метод без @Transactional, из которого вызовем метод с @Transactional, то они уже не будут работать в рамках прокси и не будут обернуты в транзакционную логику.

Что произойдет, если один метод с @Transactional вызовет другой метод с @Transactional?

Если это происходит в рамках одного сервиса, то второй транзакционный метод будет считаться частью первого, так как вызван у него изнутри, а так как спринг не знает о внутреннем вызове, то не создаст прокси для второго метода.

Что произойдет, если один метод БЕЗ @Transactional вызовет другой метод с @Transactional?

Так как Spring не знает о внутреннем вызове, то не создаст прокси для второго метода.

Будет ли транзакция отменена, если будет брошено исключение, которое указано в контракте метода?

Если в контракте описано это исключение, то она не откатится. Unchecked-исключения в транзакционном методе можно ловить, а можно и не ловить.

Расскажите про аннотации @Controller и @RestController. Чем они отличаются? Как вернуть ответ со своим статусом (например 213)?

@Controller – специальный тип класса, обрабатывает HTTP-запросы и часто используется с аннотацией @RequestMapping.

@RestController ставится на класс-контроллер вместо @Controller. Она указывает, что этот класс оперирует не моделями, а данными. Она состоит из аннотаций @Controller и @ResponseBody. Была введена в Spring 4.0 для упрощения создания RESTful веб-сервисов.

@ResponseBody сообщает контроллеру, что возвращаемый объект автоматически сериализуется (используя Jackson message converter) в json или xml и передается обратно в объект HttpResponse.

ResponseEntity используется для формирования кастомизированного HTTP-ответа с пользовательскими параметрами (заголовки, код статуса и тело ответа). Во всех остальных случаях достаточно использовать @ResponseBody.

Если хотим использовать ResponseEntity, то должны вернуть его из метода, Spring позаботится обо всем остальном.

return ResponseEntity.status(213);

Что такое ViewResolver?

ViewResolver – распознаватель представлений, это способ работы с представлениями (html-файлы), который поддерживает их распознавание на основе имени, возвращаемого контроллером.

Spring Framework поставляется с большим количеством реализаций ViewResolver. Например, класс UrlBasedViewResolver поддерживает прямое преобразование логических имен в URL.

InternalResourceViewResolver – реализация ViewResolver по умолчанию, которая позволяет находить представления, которые возвращает контроллер для последующего перехода к ним. Ищет по заданному пути, префиксу, суффиксу и имени.

Любым реализациям ViewResolver желательно поддерживать интернационализацию, то есть множество языков.

Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver) и JasperReports (JasperReportsViewResolver).

Чем отличаются Model, ModelMap и ModelAndView?

Model – интерфейс, представляет коллекцию пар ключ-значение Map<String, Object>.

Содержимое модели используется для отображения данных во View.

Например, если View выводит информацию об объекте Customer, то она может ссылаться к ключам модели, например, customerName, customerPhone, и получать значения для этих ключей.

Объекты-значения из модели также могут содержать бизнес-логику.

ModelMap – класс, наследуется от LinkedHashMap, используется для передачи значений для визуализации представления.

Преимущество ModelMap заключается в том, что он дает возможность передавать коллекцию значений и обрабатывать эти значения, как если бы они были внутри Map.

ModelAndView – это контейнер для ModelMap, объект View и HttpStatus. Это позволяет контроллеру возвращать все значения как одно.

View используется для отображения данных приложения пользователю.

Spring MVC поддерживает несколько поставщиков View (они называются шаблонизаторы) – JSP, JSF, Thymeleaf, и т. п.

Интерфейс View преобразует объекты в обычные сервлеты.

Расскажите про паттерн Front Controller, как он реализован в Spring?

Front Controller обеспечивает единую точку входа для всех входящих запросов. Все запросы обрабатываются одним обработчиком – DispatcherServlet с маппингом “/”. Этот обработчик может выполнить аутентификацию, авторизацию, регистрацию или отслеживание запроса, а затем распределяет их между контроллерами, обрабатывающими разные URL. Это и есть реализация паттерна Front Controller.

Веб-приложение может определять любое количество DispatcherServlet-ов. Каждый из них будет работать в собственном пространстве имен, загружая собственный дочерний WebApplicationContext с вьюшками, контроллерами и т. д.

- один из контекстов будет корневым, а все остальные контексты будут дочерними;
- все дочерние контексты могут получить доступ к бинам, определенным в корневом контексте, но не наоборот;
- каждый дочерний контекст внутри себя может переопределить бины из корневого контекста.

WebApplicationContext расширяет ApplicationContext (создает и управляет бинами и т. д.), но помимо этого имеет дополнительный метод getServletContext(), через который у него есть возможность получать доступ к ServletContext-у.

ContextLoaderListener создает корневой контекст приложения и будет использоваться всеми дочерними контекстами, созданными всеми DispatcherServlet-ами.

Расскажите про паттерн MVC, как он реализован в Spring?

MVC – это шаблон проектирования, делящий программу на 3 вида компонентов:

1. **Model** – модель отвечает за хранение данных.
2. **View** – отвечает за вывод данных на фронтенде.
3. **Controller** – оперирует моделями и отвечает за обмен данными model с view.

Основная цель следования принципам MVC – отделить реализацию бизнес-логики приложения (модели) от ее визуализации (view).

Spring MVC – это веб-фреймворк, основанный на Servlet API с использованием двух шаблонов проектирования Front controller и MVC.

Spring MVC реализует четкое разделение задач, что позволяет легко разрабатывать и тестировать приложения. Данные задачи разбиты между разными компонентами: Dispatcher Servlet, Controllers, View Resolvers, Views, Models, ModelAndView, Model and Session Attributes, которые полностью независимы друг от друга и отвечают только за одно направление. Поэтому MVC предоставляет большую гибкость. Он основан на интерфейсах (с классами реализации), и можно настраивать каждую часть фреймворка с помощью пользовательских интерфейсов.

Основные интерфейсы для обработки запросов:

DispatcherServlet является главным контроллером, который получает запросы и распределяет их между другими контроллерами. **@RequestsMapping** указывает, какие именно запросы будут обрабатываться в конкретном контроллере. Может быть несколько экземпляров DispatcherServlet, отвечающих за разные задачи (обработка запросов пользовательского интерфейса, REST служб и т. д.). Каждый экземпляр DispatcherServlet имеет собственную конфигурацию **WebApplicationContext**.

HandlerMapping. Выбор класса и его метода, которые должны обработать данный входящий запрос на основе любого внутреннего или внешнего для этого запроса атрибута или состояния.

Controller оперирует моделями и отвечает за обмен данными model с view.

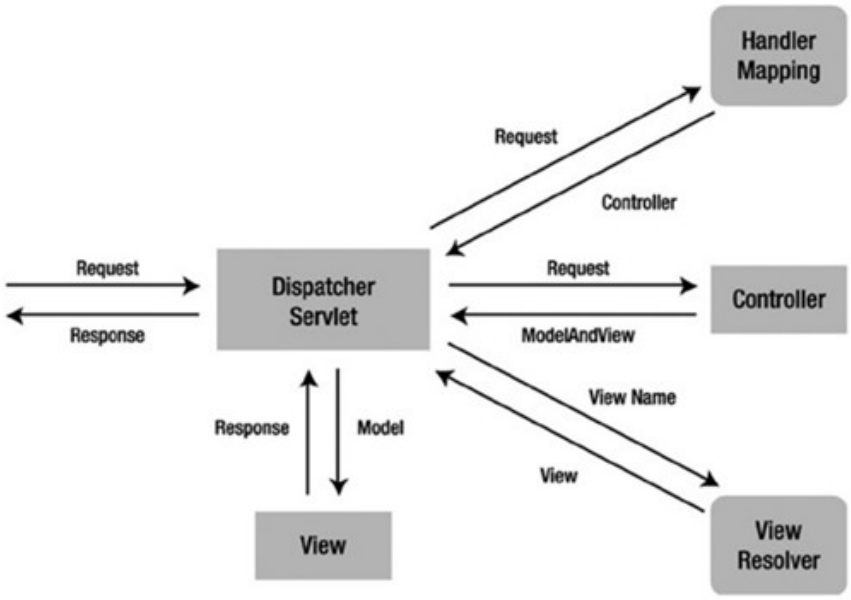
ViewResolver. Выбор, какое именно View должно быть показано клиенту на основе имени, полученного от контроллера.

View. Отвечает за возвращение ответа клиенту в виде текстов и изображений. Используются встраиваемые шаблонизаторы (Thymeleaf, FreeMarker и т. д.), так как у Spring нет родных. Некоторые запросы могут идти прямо во View, не заходя в Model, другие проходят через все слои.

HandlerAdapter. Помогает DispatcherServlet вызвать и выполнить метод для обработки входящего запроса.

ContextLoaderListener – слушатель при старте и завершении корневого класса Spring WebApplicationContext. Основным назначением является связывание жизненного цикла ApplicationContext и ServletContext, а также автоматического создания ApplicationContext. Можно использовать этот класс для доступа к бинам из различных контекстов спринга.

Ниже приведена последовательность событий, соответствующая входящему HTTP-запросу:



- после получения HTTP-запроса DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет, какой контроллер (Controller) должен быть вызван, после чего HandlerAdapter отправляет запрос в нужный метод контроллера;
- контроллер принимает запрос и вызывает соответствующий метод, вызванный метод формирует данные Model и возвращает их в DispatcherServlet вместе с именем View (как правило имя html-файла);
- при помощи интерфейса ViewResolver DispatcherServlet определяет, какое View нужно использовать на основании имени, полученного от контроллера;
- если это REST-запрос на сырые данные (JSON/XML), то DispatcherServlet сам его отправляет, минуя ViewResolver;
- если обычный запрос, то DispatcherServlet отправляет данные Model в виде атрибутов во View-шаблонизаторы Thymeleaf, FreeMarker и т. д., которые сами отправляют ответ.

Таким образом, все действия происходят через один DispatcherServlet.

Что такое АОП? Как реализовано в спринге?

Аспектно-ориентированное программирование (АОП) – это парадигма программирования, целью которой является повышение модульности за счет разделения междисциплинарных задач. Это достигается путем добавления дополнительного поведения к существующему коду без изменения самого кода.

АОП предоставляет возможность реализации сквозной логики в одном месте, т. е. логики, которая применяется к множеству частей приложения, и обеспечения автоматического применения этой логики по всему приложению.

Аспект в АОП – это модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определенных некоторым срезом.

Совет (advice) – дополнительная логика, код, который должен быть вызван из точки соединения.

Точка соединения (join point) – место в выполняемой программе (вызов метода, создание объекта, обращение к переменной), где следует применить совет.

Срез (pointcut) – набор точек соединения.

Подход Spring к АОП заключается в создании «динамических прокси» для целевых объектов и «привязывании» объектов к конфигурированному совету для выполнения сквозной логики.

Есть два варианта создания прокси-класса:

- либо он должен наследоваться от оригинального класса (CGLIB) и переопределять его методы, добавляя нужную логику;
- либо он должен имплементировать те же самые интерфейсы, что и первый класс (Dynamic Proxy).

В чем разница между Filters, Listeners and Interceptors?

Filter выполняет задачи фильтрации либо по пути запроса к ресурсу, либо по пути ответа от ресурса, либо в обоих направлениях.

Фильтры выполняют фильтрацию в методе **doFilter**. Каждый фильтр имеет доступ к объекту **FilterConfig**, из которого он может получить параметры инициализации, и ссылку на **ServletContext**. Фильтры настраиваются в дескрипторе развертывания веб-приложения.

При создании цепочки фильтров, веб-сервер решает, какой фильтр вызывать первым, в соответствии с порядком регистрации фильтров.

Когда вызывается метод doFilter(...) первого фильтра, веб-сервер создает объект FilterChain, представляющий цепочку фильтров, и передает ее в метод.

Фильтры зависят от контейнера сервлетов, могут работать с js, css.

Interceptor являются аналогом Filter в Spring. Перехватить запрос клиента можно в трех местах: preHandle, postHandle и afterCompletion.

Перехватчики работают с HandlerMapping и поэтому должны реализовывать интерфейс HandlerInterceptor или наследоваться от готового класса HandlerInterceptorAdapter, после чего переопределить указанные методы.

Чтобы добавить перехватчики в конфигурацию Spring, необходимо переопределить метод addInterceptors() внутри класса, который реализует WebMvcConfigurer.

Interceptor основан на механизме Reflection, а фильтр основан на обратном вызове функции.

preHandle – метод используется для обработки запросов, которые еще не были переданы в метод контроллера. Должен вернуть true для передачи следующему перехватчику или в handler method. False укажет на обработку запроса самим обработчиком и отсутствию необходимости передавать его дальше. Метод имеет возможность выкидывать исключения и пересылать ошибки к представлению.

postHandle – вызывается после handler method, но до обработки DispatcherServlet для передачи представлению. Может использоваться для добавления параметров в объект ModelAndView.

afterCompletion – вызывается после отрисовки представления.

Listener – это класс, имплементирующий интерфейс ServletContextListener с аннотацией **@WebListener**. Listener ждет, когда произойдет указанное событие, затем «перехватывает» событие и запускает собственное событие. Он инициализируется только один раз при запуске веб-приложения и уничтожается при остановке веб-приложения. Все ServletContextListeners уведомляются об инициализации контекста до инициализации любых фильтров или сервлетов в веб-приложении и об уничтожении контекста после того, как все сервлеты и фильтры уничтожены.

Можно ли передать в запросе один и тот же параметр несколько раз? Как?

Да, можно принять все значения, используя массив в методе контроллера:

http://localhost:8080/login?name=Ranga&name=Ravi&name=Sathish

public String method(@RequestParam(value="name") String[] names){...}

http://localhost:8080/api/foos?id=1,2,3

public String getFoos(@RequestParam List<String> id){...}

Как работает Spring Security? Как сконфигурировать? Какие интерфейсы используются?

Кратко:

Основными блоками Spring Security являются:

- **SecurityContextHolder**, чтобы обеспечить доступ к SecurityContext;
- **SecurityContext** содержит объект Authentication и в случае необходимости информацию системы безопасности, связанную с запросом;
- **Authentication** представляет принципала с точки зрения Spring Security;
- **GrantedAuthority** отражает разрешения, выданные доверителю в масштабе всего приложения;

- **UserDetails** предоставляет необходимую информацию для построения объекта Authentication из DAO-объектов приложения или других источника данных системы безопасности;
- **UserDetailsService** создает UserDetails, если передано имя пользователя в виде String (или идентификатор сертификата или что-то подобное).

Подробно:

Самым фундаментальным является **SecurityContextHolder**. В нем храним информацию о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе, работающем с приложением. По умолчанию SecurityContextHolder использует **MODE_THREADLOCAL** для хранения такой информации. Это означает, что контекст безопасности всегда доступен для методов, исполняющихся в том же самом потоке, даже если контекст безопасности явно не передается в качестве аргумента этих методов:

SecurityContextHolder.getContext().getAuthentication().getPrincipal();

UserDetails выступает в качестве принципала.

MODE_GLOBAL – все потоки Java-машины используют один контекст безопасности.

MODE_INHERITABLETHREADLOCAL – потоки, порожденные от одного защищенного потока, наличие аналогичной безопасности.

Интерфейс **UserDetailsService** – подход к загрузке информации о пользователе в Spring Security. Единственный метод этого интерфейса принимает имя пользователя в виде String и возвращает UserDetails. Он представляет собой принципала, но в расширенном виде и с учетом специфики приложения.

В случае успешной аутентификации UserDetails используется для создания Authentication объекта, который хранится в SecurityContextHolder.

Еще одним важным методом Authentication является **getAuthorities()** – массив объектов GrantedAuthority (роли).

Credentials – под ними понимаются пароль пользователя, но им может быть и отпечаток пальца, фото сетчатки и т. п.

Процесс аутентификации:

1. UsernamePasswordAuthenticationFilter получают имя пользователя и пароль и создает экземпляр класса UsernamePasswordAuthenticationToken (экземпляр интерфейса Authentication).
2. Токен передается экземпляру AuthenticationManager для проверки.
3. AuthenticationManager возвращает полностью заполненный экземпляр Authentication в случае успешной аутентификации.
4. Устанавливается контекст безопасности через вызов SecurityContextHolder.getContext().setAuthentication(...), куда передается вернувшийся экземпляр Authentication.
5. При успешной аутентификации можно использовать successHandler.

Что такое SpringBoot? Какие у него преимущества? Как конфигурируется? Подробно

Spring Boot – это модуль Spring-а, который предоставляет функцию RAD для среды Spring (Rapid Application Development – быстрая разработка приложений). Он обеспечивает более простой и быстрый способ настройки и запуска как обычных, так и веб-приложений. Он просматривает пути к классам и настроенные бины, делает разумные предположения о том, чего не хватает, и добавляет эти элементы.

Ключевые особенности и преимущества Spring Boot:

1. Простота управления зависимостями (spring-boot-starter-* в pom.xml).

Чтобы ускорить процесс управления зависимостями Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику в виде так называемых starter-пакетов.

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в приложение. Это позволяет получить универсальное решение для всех технологий, связанных со Spring, избавляя программиста от лишнего поиска необходимых зависимостей, библиотек и решения вопросов, связанных с конфликтом версий различных библиотек.

Например, если необходимо начать использовать Spring Data JPA для доступа к базе данных, можно просто включить в проект зависимость spring-boot-starter-data-jpa.

Starter-пакеты можно создавать и свои.

2. Автоматическая конфигурация.

Автоматическая конфигурация включается аннотацией **@EnableAutoConfiguration** (входит в состав аннотации **@SpringBootApplication**).

После выбора необходимых для приложения starter-пакетов Spring Boot попытается автоматически настроить Spring-приложение на основе выбранных jar-зависимостей, доступных в classpath классов, свойств в application.properties и т. п. Например, если добавим springboot-starter-web, то Spring boot автоматически сконфигурирует такие бины, как DispatcherServlet, ResourceHandlers, MessageSource и т. д.

Автоматическая конфигурация работает в последнюю очередь после регистрации пользовательских бинов и всегда отдает им приоритет. Если код уже зарегистрировал бин **DataSource**, автоконфигурация не будет его переопределять.

3. Встроенная поддержка сервера приложений/контейнера сервлетов (**Tomcat, Jetty**).

Каждое Spring Boot web-приложение включает встроенный web-сервер. Не нужно беспокоиться о настройке контейнера сервлетов и развертывания приложения в нем. Теперь приложение может запускаться само как исполняемый .jar-файл с использованием встроенного сервера.

4. Готовые к работе функции, такие как метрики, проверки работоспособности, security и внешняя конфигурация.

5. Инструмент CLI (command-line interface) для разработки и тестирования приложения Spring Boot.

6. Минимизация boilerplate кода (код, который должен быть включен во многих местах практически без изменений), конфигурации XML и аннотаций.

Как происходит автоконфигурация в Spring Boot:

1. Отмечаем main-класс аннотацией **@SpringBootApplication** (аннотация инкапсулирует в себя: **@SpringBootConfiguration, @ComponentScan, @EnableAutoConfiguration**), таким образом наличие @SpringBootApplication включает сканирование компонентов, автоконфигурацию и показывает разным компонентам Spring (например, интеграционным тестам), что это приложение Spring Boot.
2. @EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector. Этот класс не объявляет бины сам, а использует фабрики.
3. Класс EnableAutoConfigurationImportSelector импортирует ВСЕ (более 150) перечисленные в META-INF/spring.factories конфигурации, чтобы предоставить нужные бины в контекст приложения.
4. Каждая из этих конфигураций пытается сконфигурировать различные аспекты приложения (web, JPA, AMQP и т. д.), регистрируя нужные бины. Логика при регистрации бинов управляется набором @ConditionalOn* аннотаций. Можно указать, чтобы бин создавался при наличии класса в classpath (@ConditionalOnClass), наличии существующего бина (@ConditionalOnBean), отсутствии бина (@ConditionalOnMissingBean) и т. п. Таким образом, наличие конфигурации не значит, что бин будет создан и зачастую конфигурация ничего делать и создавать не будет.
5. Созданный в итоге AnnotationConfigEmbeddedWebApplicationContext ищет в том же DI- контейнере фабрику для запуска embedded servlet container.
6. Servlet container запускается, приложение готово к работе.

Расскажите про нововведения Spring 5

- используется JDK 8+ (Optional, CompletableFuture, Time API, java.util.function, default methods);
- поддержка Java 9 (Automatic-Module-Name in 5.0, module-info in 6.0+, ASM 6);
- поддержка HTTP/2 (TLS, Push), NIO/NIO.2;
- поддержка Kotlin;
- реактивность (веб-инфраструктура с реактивным стеком, «Spring WebFlux»);
- Null-safety аннотации(@Nullable), новая документация;
- совместимость с Java EE 8 (Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0);
- поддержка JUnit 5 + Testing Improvements (conditional and concurrent);
- удалена поддержка Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava.

Паттерны

Что такое «шаблон проектирования»?

Проверенное и готовое к использованию логическое решение, которое может быть реализовано по-разному в разных языках программирования.

Плюсы:

- снижение сложности разработки за счет готовых абстракций;
- облегчение коммуникации между разработчиками.

Минусы:

- слепое следование некоторому шаблону может привести к усложнению программы;
- желание попробовать некоторый шаблон в деле без особых на то оснований.

Назовите основные характеристики шаблонов

- **имя** – все шаблоны имеют уникальное имя, служащее для их идентификации;
- **назначение** данного шаблона;
- **задача**, которую шаблон позволяет решить;
- **способ решения**, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден;
- **участники** – сущности, принимающие участие в решении задачи;
- **следствия** от использования шаблона как результат действий, выполняемых в шаблоне;
- **реализация** – возможный вариант реализации шаблона.

Назовите три основные группы паттернов

Порождающие – отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов без внесения в программу лишних зависимостей.

Структурные – отвечают за построение удобных в поддержке иерархий классов.

Поведенческие – заботятся об эффективной коммуникации между объектами.

Основные – основные строительные блоки, используемые для построения других шаблонов. Например, интерфейс.

Расскажите про паттерн «Одиночка» (Singleton)

Порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Конструктор помечается как private, а для создания нового объекта Singleton использует специальный метод getInstance(). Он либо создает объект, либо отдаёе существующий объект, если он уже был создан.

private static Singleton instance;

```
public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
            }
    return instance;
  }
```

Плюсы:

- можно не создавать множество объектов для ресурсоемких задач, а пользоваться одним.

Минусы:

- нарушает принцип единой ответственности, так как его могут использовать множество объектов.

Почему считается антипаттерном?

- нельзя тестировать с помощью mock, но можно использовать powerMock;
- нарушает принцип единой ответственности;
- нарушает Open/Close принцип, его нельзя расширить.

Можно ли его синхронизировать без synchronized у метода?

1. Можно сделать его Enum (eager). Это статический final класс с константами. JVM загружает final и static классы на этапе компиляции, а значит несколько потоков не могут создать несколько инстансов.
2. С помощью double checked locking (lazy). Synchronized внутри метода:

```
private static volatile Singleton instance;
public static Singleton getInstance() {
        Singleton localInstance = instance;
        if (localInstance == null) {                    // first check
            synchronized (Singleton.class) {
                localInstance = instance;
                if (localInstance == null) {      // second check
                    instance = localInstance = new Singleton();
                }
            }
        }
        return localInstance;
    }
```

Расскажите про паттерн «Строитель» (Builder)

Порождающий паттерн, который позволяет создавать сложные объекты пошагово. Строитель дает возможность использовать один и тот же код для получения разных представлений одного объекта.

Паттерн предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.

Процесс конструирования объекта разбит на отдельные шаги (например, построитьСтены, вставитьДвери). Чтобы создать объект, нужно поочередно вызывать методы строителя. Причем не нужно запускать все шаги, а только те, что нужны для производства объекта определенной конфигурации.

Можно пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый Директором. В этом случае Директор будет задавать порядок шагов строительства, а строитель – выполнять их.

Плюсы:

- позволяет использовать один и тот же код для создания различных объектов;
- изолирует сложный код сборки объектов от его основной бизнес-логики.

Минусы:

- усложняет код программы из-за введения дополнительных классов.

Расскажите про паттерн «Фабричный метод» (Factory Method)

Порождающий шаблон проектирования, в котором подклассы имплементируют общий интерфейс с методом для создания объектов. Переопределенный метод в каждом наследнике возвращает нужный вариант объекта.

Объекты все равно будут создаваться при помощи **new**, но делать это будет фабричный метод. Таким образом можно переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.

Плюсы:

- выделяет код производства объектов в одно место, упрощая поддержку кода;
- реализует принцип открытости/закрытости.

Минусы:

- может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Пример: **SessionFactory** в Hibernate.

Расскажите про паттерн «Абстрактная фабрика» (Abstract Factory)

Порождающий паттерн проектирования, который представляет собой интерфейс для создания других классов, не привязываясь к конкретным классам создаваемых объектов.

Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.

Далее создается абстрактная фабрика – общий интерфейс, который содержит фабричные методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые выделили ранее – Кресла, Диваны и Столики.

Плюсы:

- гарантированно будет создаваться тип одного семейства.

Минусы:

- усложняет код программы из-за введения множества дополнительных классов.

Расскажите про паттерн «Прототип» (Prototype)

Порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Паттерн поручает создание копий самим копируемым объектам. Он вводит общий интерфейс с методом **clone** для всех объектов, поддерживающих клонирование. Реализация этого метода в разных классах очень схожа. Метод создает новый объект текущего класса и копирует в него значения всех полей собственного объекта.

Плюсы:

- позволяет клонировать объекты, не привязываясь к их конкретным классам.

Минусы:

- сложно клонировать составные объекты, имеющие ссылки на другие объекты.

Расскажите про паттерн «Адаптер» (Adapter)

Структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов так, что другой объект даже не знает о наличии первого.

Плюсы:

- отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Минусы:

- усложняет код программы из-за введения дополнительных классов.

Расскажите про паттерн «Декоратор» (Decorator)

Структурный паттерн проектирования, который позволяет добавлять объектам новую функциональность, оборачивая их в полезные «обертки».

Целевой объект помещается в другой объект-обертку, который запускает базовое поведение обернутого объекта, а затем добавляет к результату что-то свое.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать – чистым или обернутым. Можно использовать несколько разных оберток одновременно – результат будет иметь объединенное поведение всех оберток сразу.

Адаптер не меняет состояния объекта, а декоратор может менять.

Плюсы:

- большая гибкость, чем у наследования.

Минусы:

- труднее конфигурировать многократно обернутые объекты.

Расскажите про паттерн «Заместитель» (Proxy)

Структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители, которые перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

Заместитель предлагает создать новый класс-дублер, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта, выполняя промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте.

Плюсы:

- позволяет контролировать сервисный объект незаметно для клиента.

Минусы:

- увеличивает время отклика от сервиса.

Расскажите про паттерн «Итератор» (Iterator)

Поведенческий паттерн проектирования, который дает возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Идея состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

Детали: создается итератор и интерфейс, который возвращает итератор. В классе, в котором надо будет вызывать итератор, имплементируем интерфейс, возвращающий итератор, а сам итератор делаем там нестатическим вложенным классом, так как он нигде использоваться больше не будет.

Расскажите про паттерн «Шаблонный метод» (Template Method)

Поведенческий паттерн проектирования, который пошагово определяет алгоритм и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

Паттерн предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном шаблонном методе друг за другом. Для описания шагов используется абстрактный класс. Общие шаги можно будет описать прямо в абстрактном классе. Это позволит подклассам переопределять некоторые шаги алгоритма,

оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.

Расскажите про паттерн «Цепочка обязанностей» (Chain of Responsibility)

Поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Базируется на том, чтобы превратить каждую проверку в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы.

Каждый из методов будет иметь ссылку на следующий метод-обработчик, что образует цепь. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке. Может и не передавать, если проверка в одном из методов не прошла.

Какие паттерны используются в Spring Framework?

- Singleton – Bean scopes;
- Factory – Bean Factory classes;
- Prototype – Bean scopes;
- Adapter – Spring Web and Spring MVC;
- Proxy – Spring Aspect Oriented Programming support;
- Template Method – JdbcTemplate, HibernateTemplate etc;
- Front Controller – Spring MVC DispatcherServlet;
- DAO – Spring Data Access Object support;
- Dependency Injection.

Какие паттерны используются в Hibernate?

- Domain Model – объектная модель предметной области, включающая в себя как поведение, так и данные;
- Data Mapper – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя;
- Proxy – применяется для ленивой загрузки;
- Factory – используется в SessionFactory.

Шаблоны GRASP: Low Coupling (низкая связанность) и High Cohesion (высокая сплоченность)

Low Coupling – части системы, которые изменяются вместе, должны находиться близко друг к другу.

High Cohesion – если возвести Low Coupling в абсолют, то можно прийти к тому, чтобы разместить всю функциональность в одном единственном классе. В таком случае связей не будет вообще, но в этот класс попадет совершенно несвязанная между собой бизнес-логика.

Принцип High Cohesion говорит следующее: части системы, которые изменяются параллельно, должны иметь как можно меньше зависимостей друг на друга.

Low Coupling и High Cohesion представляют из себя два связанных между собой паттерна, рассматривать которые имеет смысл только вместе. Их суть: система должна состоять из слабо связанных классов, которые содержат связанную бизнес-логику. Соблюдение этих принципов позволяет удобно переиспользовать созданные классы, не теряя понимания о их зоне ответственности.

Расскажите про паттерн Saga

Сага – это механизм, обеспечивающий согласованность данных в микросервисах без применения распределенных транзакций.

Для каждой системной команды, которой надо обновлять данные в нескольких сервисах, создается некоторая сага. Сага представляет из себя некоторый «чек-лист», состоящий из последовательных локальных ACID-транзакций, каждая из которых обновляет данные в одном сервисе. Для обработки сбоев применяется компенсирующая транзакция. Такие транзакции выполняются в случае сбоя на всех сервисах, на которых локальные транзакции выполнились успешно.

Типов транзакций в саге четыре:

- **компенсирующая** – отменяет изменение, сделанное локальной транзакцией;
- **компенсируемая** – это транзакция, которую необходимо компенсировать (отменить) в случае, если последующие транзакции завершаются неудачей;
- **поворотная** – транзакция, определяющая успешность всей саги: если она выполняется успешно, то сага гарантированно дойдет до конца;
- **повторяемая** – идет после поворотной и гарантированно завершается успехом.

Алгоритмы

Что такое Big O? Как происходит оценка асимптотической сложности алгоритмов?

Big O (O большое / символ Ландау) – математическое обозначение порядка функции для сравнения асимптотического поведения функций.

Асимптотика – характер изменения функции при стремлении ее аргумента к определенной точке.

Любой алгоритм состоит из неделимых операций процессора (шагов), поэтому вместо секунд нужно измерять время в операциях процессора.

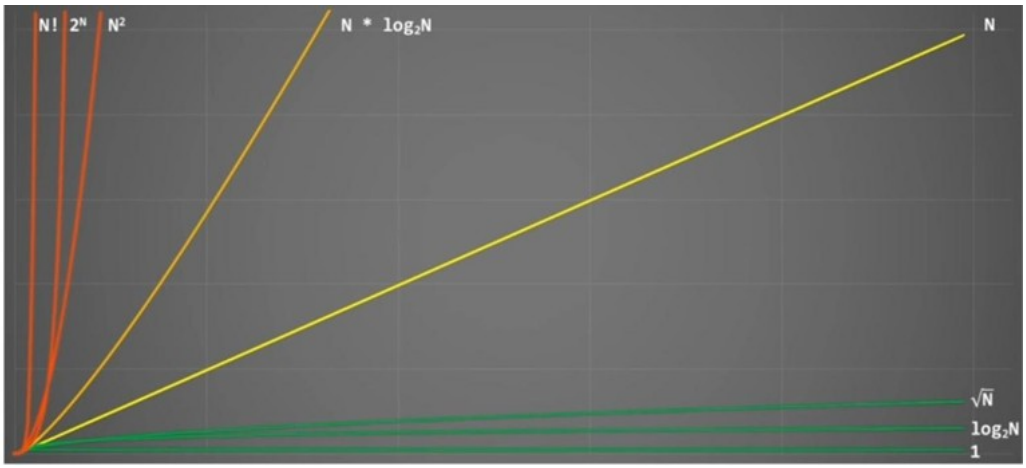
DTIME – количество шагов (операций процессора), необходимых, чтобы алгоритм завершился.

Временная сложность обычно оценивается путем подсчета числа элементарных операций, осуществляемых алгоритмом. Время исполнения одной такой операции при этом берется константой, то есть асимптотически оценивается как O(1).

Сложность алгоритма состоит из двух факторов: временная сложность и сложность по памяти. Временная сложность – функция, представляющая зависимость количество операций процессора, необходимых, чтобы алгоритм завершился, от размера входных данных. Все неделимые операции языка (операции сравнения, арифметические, логические, инициализации и возврата) считаются выполняемыми за 1 операцию процессора, эта погрешность считается приемлемой. При росте N, слагаемые с меньшей скоростью роста все меньше влияют на значение функции. Поэтому вне зависимости от констант при слагаемых слагаемое с большей скорость роста определяет значение функции. Данное слагаемое называют порядком функции.

Пример: T(N) = 5 * N^2 + 999 * N... Где (5 * N^2) и (9999 * N) являются слагаемыми функции. Константы (5 и 999) не указываются в рамках нотации Big O, так как не показывают абсолютную сложность алгоритма, поскольку могут изменяться в зависимости от машины, поэтому сложность равна O(N^2).

Функции по убыванию их порядка
$N!$
2^N
N^2
$N * \log_2 N$
N
$\sqrt{N}$
$\log_2 N$
1



<u>В порядке возрастания сложности:</u>

1. O(1) – константная, чтение по индексу из массива.

2. O(log(n)) – логарифмическая, бинарный поиск в отсортированном массиве.

3. O(√n) – сублинейная.

4. O(n) – линейная, перебор массива в цикле, два цикла подряд, линейный поиск наименьшего или наибольшего элемента в неотсортированном массиве.

5. O(n*log(n)) – квазилинейная, сортировка слиянием, сортировка кучей.

6. O(n^2) – полиномиальная (квадратичная), вложенный цикл, перебор двумерного массива, сортировка пузырьком, сортировка вставками.

7. O(2^n) – экспоненциальная, алгоритмы разложения на множители целых чисел.

8. O(n!) - факториальная, решение задачи коммивояжёра полным перебором

Алгоритм считается приемлемым, если сложность не превышает O(n*log(n)), иначе говнокод.

n – количество операций.

Что такое рекурсия? Сравните преимущества и недостатки итеративных и рекурсивных алгоритмов (с примерами)

Рекурсия – способ отображения какого-либо процесса внутри самого этого процесса, то есть ситуация, когда процесс является частью самого себя.

Рекурсия состоит из базового случая и шага рекурсии. Базовый случай представляет собой самую простую задачу, которая решается за одну итерацию, например, if(n == 0) return 1.

В базовом случае обязательно присутствует условие выхода из рекурсии.

Смысл рекурсии в движении от исходной задачи к базовому случаю, пошагово уменьшая размер исходной задачи на каждом шаге рекурсии.

После того как будет найден базовый случай, срабатывает условие выхода из рекурсии, и стек рекурсивных вызовов разворачивается в обратном порядке, пересчитывая результат исходной задачи, который основан на результате, найденном в базовом случае.

Так работает рекурсивное вычисление факториала:

```
int factorial(int n) {
    if(n == 0) return 1;                // базовый случай с условием выхода
    else return n * factorial(n - 1); // шаг рекурсии (рекурсивный вызов)
  }
```

Или так:

```
return (n==0) ? 1 : n * factorial(n-1);
```

Рекурсия имеет линейную сложность O(n).

Циклы дают лучшую производительность, чем рекурсивные вызовы, поскольку вызовы методов потребляют больше ресурсов, чем исполнение обычных операторов.

Циклы гарантируют отсутствие переполнения стека, т. к. не требуется выделения дополнительной памяти.

В случае рекурсии стек вызовов разрастается и его необходимо просматривать для получения конечного ответа.

При использовании головной рекурсии необходимо принимать во внимание размер стека.

Если уровней вложенности много или изменятся, то предпочтительна рекурсия. Если их несколько, то лучше цикл.

Что такое жадные алгоритмы? Приведите пример

Жадные алгоритмы являются одной из трех техник создания алгоритмов, вместе с принципом «Разделяй и властвуй» и динамическим программированием.

Жадный алгоритм – это алгоритм, который на каждом шагу совершает локально оптимальные решения, т. е. максимально возможное из допустимых, не учитывая предыдущие или следующие шаги. Последовательность этих локально оптимальных решений приводит (не всегда) к глобально оптимальному решению.

Т. е. задача разбивается на подзадачи, в каждой подзадаче делается оптимальное решение, и в итоге вся задача решается оптимально. При этом важно является ли каждое локальное решение безопасным шагом. **Безопасный шаг** – это шаг, приводящий к оптимальному решению.

К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе жадный, потому что на каждом шагу ищем вершину с наименьшим весом, в которой еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.

Пример: наименьшая яма с кладом.

Расскажите про пузырьковую сортировку

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован.

Асимптотика в худшем и среднем случае – O(n^2), в лучшем случае – O(n) – массив уже отсортирован.

Расскажите про быструю сортировку

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Для этого используются дополнительные переменные – значения слева и справа, которые сравниваются с опорным элементом. Рекурсивно вызовемся от каждой из частей, где будет выбран новый опорный элемент. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного.

Асимптотика: O(n*log(n)) в среднем и лучшем случае. Наихудшая оценка O(n^2) достигается при неудачном выборе опорного элемента.

Расскажите про сортировку слиянием

Основана на парадигме «разделяй и властвуй». Будем делись массив пополам, пока не получим множество массивов из одного элемента. После чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Так сделаем слияния массивов из первого элемента в массивы по 2 элемента, затем из двух в 4 и т. д.

Слияние работает за O(n), уровней всего log(n), поэтому асимптотика O(n*log(n)).

Расскажите про бинарное дерево

Бинарное дерево – иерархическая структура данных, в которой каждый узел может иметь двух потомков. Как правило, первый называется родительским узлом, а наследники называются левым и правым нодами/узлами. Каждый узел в дереве задает поддерево, корнем которого он является. Оба поддерева – левое и правое – тоже являются бинарными деревьями. Ноды, которые не имеют потомков, называются листьями дерева. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X. Этим достигается упорядоченная структура данных, то есть всегда отсортированная.

Поиск в лучшем случае – O(log(n)), худшем – O(n) при вырождении в связанный список.

Расскажите про красно-черное дерево

Усовершенствованная версия бинарного дерева. Каждый узел в красно-черном дереве имеет дополнительное поле цвет. Красно-черное дерево отвечает следующим требованиям:

- узел либо красный, либо черный;
- корень черный;
- все листья черные и не хранят данных;
- оба потомка каждого красного узла черные;
- любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов; если не одинаковое, то происходит переворот.

При добавлении постоянно увеличивающихся/уменьшающихся чисел в бинарное дерево оно вырождается в связанный список и теряет свои преимущества. Тогда как красно-черное дерево может потребовать до двух поворотов для поддержки сбалансированности, чтобы избежать вырождения.

При операциях удаления в бинарном дереве для удаляемого узла надо найти замену. Красно-черное дерево сделает то же самое, но потребует до трех поворотов для поддержки сбалансированности.

В этом и состоит преимущество.

Сложность поиска, вставки и удаления – O(log(n)).

Расскажите про линейный и бинарный поиск

Линейный поиск – сложность O(n), так как все элементы проверяются по очереди.

Бинарный поиск – O(log(n)). Массив должен быть отсортирован. Происходит поиск индекса в массиве, содержащего искомое значение.

1. Берем значение из середины массива и сравниваем с искомым. Индекс середины считается по формуле mid = (high + low) / 2.

low – индекс начала левого подмассива;

high – индекс конца правого подмассива.

2. Если значение в середине больше искомого, то рассматриваем левый подмассив и high = middle - 1.

3. Если меньше, то правый и low = middle + 1.

4. Повторяем, пока mid не становится равен искомому элементу или подмассив не станет пустым.

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high)/2;

        if (key > a[mid]) {
            low = mid + 1;
        } else if (key < a[mid]) {
            high = mid - 1;
        } else return mid;
    }
    return -1;
}
```

Расскажите про очередь и стек

Stack – это область хранения данных, находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок-фрейм, который содержит локальные переменные метода и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок 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.

Queue – это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out), поэтому 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. 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.

Сравните сложность вставки, удаления, поиска и доступа по индексу в ArrayList и LinkedList

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

HashSet – временная сложность основных операций			
	Поиск	**Вставка**	**Удаление**
Метод	**contains(object)**	**add(object)**	**remove(index)**
Среднее	O(1)	O(1)	O(1)
Худшее (до Java 8)	O(n)	O(n)	O(n)
Худшее (Java 8+)	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

LinkedHashSet – временная сложность основных операций			
	Поиск	**Вставка**	**Удаление**
Метод	**contains(object)**	**add(object)**	**remove(index)**
Среднее	O(1)	O(1)	O(1)
Худшее (до Java 8)	O(n)	O(n)	O(n)
Худшее (Java 8+)	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

TreeSet – временная сложность основных операций			
	Поиск	**Вставка**	**Удаление**
Метод	**contains(object)**	**add(object)**	**remove(index)**
Среднее	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$
Худшее	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

ArrayList – временная сложность основных операций				
	Индекс	**Поиск**	**Вставка**	**Удаление**
Метод	**get(i)**	**contains(object)**	**add(object)**	**remove(index)**
Среднее	O(1)	O(n)	O(n)	O(n)
Худшее	O(1)	O(n)	O(n)	O(n)