Вопросы и ответы к интервью JAVA разработчика

ООП

Что такое ООП?

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

— объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы;
— каждый объект является экземпляром определенного класса
— классы образуют иерархии.

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

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

Назовите основные принципы ООП.

Инкапсуляция - сокрытие реализации.
Наследование - создание новой сущности на базе уже существующей.
Полиморфизм - возможность иметь разные формы для одной и той же сущности.
Абстракция - набор общих характеристик.
Посылка сообщений - форма связи, взаимодействия между сущностями.
Переиспользование- все что перечислено выше работает на повторное использование кода.

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

Что такое «инкапсуляция»?

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

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

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

Теперь вернёмся в сегодняшний день к современным чудесам автопрома с коробкой-автоматом.

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

Пример:

```
public class SomePhone {

   private int year;
   private String company;
   public SomePhone(int year, String company) {
      this.year = year;
      this.company = company;
   }
   private void openConnection(){
      //findComutator
      //openNewConnection...
   }
   public void call() {
      openConnection();
      System.out.println(«Вызываю номер»);
   }

   public void ring() {
      System.out.println(«Дзынь-дзынь»);
   }

}
```

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

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

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

Что такое «наследование»?

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

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

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

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

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

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

```
public abstract class WirelessPhone extends AbstractPhone {

  private int hour;

  public WirelessPhone(int year, int hour) {
    super(year);
    this.hour = hour;
  }
}
```

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

```
public class CellPhone extends WirelessPhone {
  public CellPhone(int year, int hour) {
    super(year, hour);
  }

  @Override
  public void call(int outputNumber) {
    System.out.println(«Вызываю номер « + outputNumber);
  }

  @Override
  public void ring(int inputNumber) {
    System.out.println(«Вам звонит абонент « + inputNumber);
  }
}
```

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

кода класс можно описать так:

```
public class Smartphone extends CellPhone {

   private String operationSystem;

   public Smartphone(int year, int hour, String operationSystem) {
      super(year, hour);
      this.operationSystem = operationSystem;
   }

   public void install(String program){
      System.out.println(«Устанавливаю « + program + «для» + operationSystem);
   }

}
```

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

Что такое «полиморфизм»?

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

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

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

Всё дело в том, что основные элементы управления автомобиля имеют одну и ту же конструкцию, и принцип действия. Водитель точно знает, что для того, чтобы повернуть налево, он должен повернуть руль, независимо от того, есть там гидроусилитель или нет. Если человеку надо доехать с работы до дома, то он сядет за руль автомобиля и будет выполнять одни и те же действия, независимо от того, какой именно тип автомобиля он использует. По сути, можно сказать, что все автомобили имеют один и тот же интерфейс, а водитель, абстрагируясь от сущности автомобиля, работает именно с этим интерфейсом. Если водителю предстоит ехать по немецкому автобану, он, вероятно выберет быстрый автомобиль с низкой посадкой, а если предстоит возвращаться из отдалённого маральника в Горном Алтае после дождя, скорее всего, будет выбран УАЗ с армейскими мостами. Но, независимо от того, каким образом будет реализовываться движение и внутреннее функционирование машины, интерфейс останется прежним.
Полиморфная переменная, это переменная, которая может принимать значения разных типов, а полиморфная функция, это функция у которой хотя бы один аргумент является полиморфной переменной. Выделяют два вида полиморфных функций:
ad hoc, функция ведет себя по разному для разных типов аргументов (например, функция draw()

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

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

Пример:

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

```
public class User {
   private String name;

   public User(String name) {
      this.name = name;
   }

   public void callAnotherUser(int number, AbstractPhone phone) {
// вот он полиморфизм - использование в коде абстактного типа AbstractPhone phone!
      phone.call(number);
   }
}
```

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

```
public class ThomasEdisonPhone extends AbstractPhone {

   public ThomasEdisonPhone(int year) {
      super(year);
   }

   @Override
   public void call(int outputNumber) {
      System.out.println(«Вращайте ручку»);
      System.out.println(«Сообщите номер абонента, сэр»);
   }

   @Override
   public void ring(int inputNumber) {
      System.out.println(«Телефон звонит»);
   }
}
```

Обычный стационарный телефон:

```
public class Phone extends AbstractPhone {

   public Phone(int year) {
      super(year);
   }
```

```
    @Override
    public void call(int outputNumber) {
        System.out.println(«Вызываю номер» + outputNumber);
    }

    @Override
    public void ring(int inputNumber) {
        System.out.println(«Телефон звонит»);
    }
}
```

И, наконец, крутой видеотелефон:

```
public class VideoPhone extends AbstractPhone {

    public VideoPhone(int year) {
        super(year);
    }

    @Override
    public void call(int outputNumber) {
        System.out.println(«Подключаю видеоканал для абонента « + outputNumber);
    }

    @Override
    public void ring(int inputNumber) {
        System.out.println(«У вас входящий видеовызов...» + inputNumber);
    }
}
```

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

```
AbstractPhone firstPhone = new ThomasEdisonPhone(1879);
AbstractPhone phone = new Phone(1984);
AbstractPhone videoPhone=new VideoPhone(2018);
User user = new User(«Андрей»);
user.callAnotherUser(224466,firstPhone);
// Вращайте ручку
//Сообщите номер абонента, сэр
user.callAnotherUser(224466,phone);
//Вызываю номер 224466
user.callAnotherUser(224466,videoPhone);
//Подключаю видеоканал для абонента 224466
```

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

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

Обычно, при переопределении метода, используется аннотация @Override, которая подсказывает

компилятору о необходимости проверить сигнатуры переопределяемого и переопределяющего методов.

Что такое «абстракция»?

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

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

Пример:

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();

  // Regular method
  public void sleep() {
    System.out.println(«Zzz»);
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println(«The pig says: wee wee»);
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Что представляет собой «обмен сообщениями»?

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

Пусть требуется создать физическую модель, описывающую сталкивающиеся шары разных раз-

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

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

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

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

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

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

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

к оглавлению

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

Преимущества:

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

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

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

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

Возможность создавать расширяемые системы.

Использование полиморфизма оказывается полезным при:

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

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

того, какой используется объект.

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

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

Повторное использование кода:

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

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

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

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

Недостатки:

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

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

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

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

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

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

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

В чем разница между композицией и агрегацией?

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

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

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

Что такое статическое и динамическое связывание?

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

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

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

JVM

За что отвечает JVM:

Загрузка, проверка и исполнение байт кода;
Предоставление среды выполнения для выполнения байт-кода;
Управление памятью и очисткой мусора (Garbage collection);

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

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

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

Типы ссылок и значения

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

Classloader

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

При запуске JVM, используются три загрузчика классов:

Bootstrap class loader (Загрузчик класса Bootstrap)
Extensions class loader (Загрузчик класса расширений)
System class loader (Системный загрузчик классов)

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

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

Системный загрузчик загружает код, найденный в java.class.path, который сопоставляется с пере-

менной среды CLASSPATH. Это реализуется классом sun.misc.Launcher$AppClassLoader.

Загрузчик классов выполняет три основных действия в строгом порядке:

Загрузка: находит и импортирует двоичные данные для типа.
Связывание: выполняет проверку, подготовку и (необязательно) разрешение.
Проверка: обеспечивает правильность импортируемого типа.
Подготовка: выделяет память для переменных класса и инициализация памяти значениями по умолчанию.
Разрешение: преобразует символические ссылки из типа в прямые ссылки.
Инициализация: вызывает код Java, который инициализирует переменные класса их правильными начальными значениями.

Пользовательский загрузчик классов

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

Что делает возможным следующее:

загружать или выгружать классы во время выполнения (например, динамически загружать библиотеки во время выполнения, даже из ресурса HTTP). Это важная особенность для:
реализация скриптовых языков;
использование bean builders;
добавить пользовательскую расширение;
позволяя нескольким пространствам имен общаться. Например, это одна из основ протоколов CORBA / RMI;
изменить способ загрузки байт-кода (например, можно использовать зашифрованный байт-код класса Java);
модифицировать загруженный байт-код (например, для переплетения аспектов во время загрузки при использовании аспектно-ориентированного программирования);

Области данных времени выполнения

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

The pc Register (PCR)

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

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

Java Virtual Machine Stacks

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

Heap

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

Method Area

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

Run-Time Constant Pool

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

Native Method Stacks

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

Frames

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

метода.

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

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

Локальные переменные

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

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

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

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

Стеки операндов (Operand Stacks)

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

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

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

Динамическое связывание (Dynamic Linking)

Каждый frame содержит ссылку на run-time constant pool для типа текущего метода для поддержки

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

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

Нормальное завершение вызова метода

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

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

Резкое завершение вызова метода

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

Execution Engine

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

Interpreter

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

JIT Compiler

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

Генератор промежуточного кода (Intermediate Code Generator). Производит промежуточный код. Code Optimizer. Отвечает за оптимизацию промежуточного кода, сгенерированного выше.

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

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

Garbage Collector

JAVA Core

Чем различаются JRE, JVM и JDK?

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

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

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

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

Какие существуют модификаторы доступа?

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

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

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

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

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

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

О чем говорит ключевое слово final?

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

- Класс не может иметь наследников;
- Метод не может быть переопределен в классах наследниках;
- Поле не может изменить свое значение после инициализации;
- Параметры методов не могут изменять своё значение внутри метода;
- Локальные переменные не могут быть изменены после присвоения им значения.

Какими значениями инициализируются переменные по умолчанию?

Числа инициализируются 0 или 0.0;
char — \u0000;
boolean — false;
Объекты (в том числе String) — null.

Что вы знаете о функции main()?

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

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

Какие логические операции и операторы вы знаете?

&: Логическое AND (И);
&&: Сокращённое AND;
|: Логическое OR (ИЛИ);
||: Сокращённое OR;
^: Логическое XOR (исключающее OR (ИЛИ));
!: Логическое унарное NOT (НЕ);
&=: AND с присваиванием;
|=: OR с присваиванием;
^=: XOR с присваиванием;
==: Равно;
!=: Не равно;
?:: Тернарный (троичный) условный оператор.

Что такое тернарный оператор выбора?

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

Выражение записывается в следующей форме:

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

Какие побитовые операции вы знаете?

~: Побитовый унарный оператор NOT;
&: Побитовый AND;
&=: Побитовый AND с присваиванием;
|: Побитовый OR;
|=: Побитовый OR с присваиванием;

^: Побитовый исключающее XOR;
^=: Побитовый исключающее XOR с присваиванием;
>>: Сдвиг вправо (деление на 2 в степени сдвига);
>>=: Сдвиг вправо с присваиванием;
>>>: Сдвиг вправо без учёта знака;
>>>=: Сдвиг вправо без учёта знака с присваиванием;
<<: Сдвиг влево (умножение на 2 в степени сдвига);
<<=: Сдвиг влево с присваиванием.

Где и для чего используется модификатор abstract?

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

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

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

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

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

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

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

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

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

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

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

Интерфейсы позволяют создавать структуры типов без иерархии.

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

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

Почему в некоторых интерфейсах вообще не определяют методов?

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

Почему нельзя объявить метод интерфейса с модификатором final?

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

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

Интерфейс.

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

Внутри класса доступ к приватной переменной открыт без ограничений;

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

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

Через механизм рефлексии (Reflection API):

```
class Victim {
  private int field = 42;
}
//...
Victim victim = new Victim();
Field field = Victim.class.getDeclaredField(«field»);
field.setAccessible(true);
int fieldValue = (int) field.get(victim);
//...
```

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

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

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

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

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

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

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

Пример 1:

```
public class MainClass {

  public static void main(String args[]) {
    System.out.println(TestClass.v);
    new TestClass().a();
  }

}

public class TestClass {

  public static String v = «Some val»;

  {
    System.out.println(«!!! Non-static initializer»);
  }

  static {
    System.out.println(«!!! Static initializer»);
  }

  public void a() {
    System.out.println(«!!! a() called»);
  }

}
```

Результат выполнения:

```
!!! Static initializer
Some val
!!! Non-static initializer
!!! a() called
```

Пример 2:

```
public class MainClass {

   public static void main(String args[]) {
      new TestClass().a();
   }

}

public class TestClass {

   public static String v = «Some val»;

   {
      System.out.println(«!!! Non-static initializer»);
   }

   static {
      System.out.println(«!!! Static initializer»);
   }

   public void a() {
      System.out.println(«!!! a() called»);
   }

}
```

Результат выполнения:

```
!!! Static initializer
!!! Non-static initializer
!!! a() called
```

Зачем нужны и какие бывают блоки инициализации?

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

Существуют статические и нестатические блоки инициализации.

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

Несколько блоков инициализации выполняются в порядке следования в коде класса.

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

Блок инициализации возможно создать и в анонимном классе.

К каким конструкциям Java применим модификатор static?

- полям;
- методам;
- вложенным классам;
- членам секции import.

Для чего в Java используются статические блоки инициализации?

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

Что произойдёт, если в блоке инициализации возникнет исключительная ситуация?

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

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

Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?

Если возникшее исключение - наследник RuntimeException:

- для статических блоков инициализации будет выброшено java.lang.ExceptionInInitializerError;
- для нестатических будет проброшено исключение-источник.

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

Может ли статический метод быть переопределён или перегружен?

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

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

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

Могут ли нестатические методы перегрузить статические?

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

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

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

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

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

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

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

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

Как получить доступ к переопределенным методам родительского класса?

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

```
super.method();
```

Можно ли объявить метод абстрактным и статическим одновременно?

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

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

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

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

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

Пример:

```
public class MainClass {

	public static void main(String args[]) {
		System.out.println(TestClass.v);
		new TestClass().a();
		System.out.println(TestClass.v);
	}

}

public class TestClass {

	public static String v = «Initial val»;

	{
		System.out.println(«!!! Non-static initializer»);
		v = «Val from non-static»;
	}

	static {
		System.out.println(«!!! Static initializer»);
		v = «Some val»;
	}

	public void a() {
		System.out.println(«!!! a() called»);
	}

}
```

Результат:

```
!!! Static initializer
Some val
!!! Non-static initializer
!!! a() called
Val from non-static
```

Где разрешена инициализация статических/нестатических полей?

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

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

Какие типы классов бывают в java?

- Top level class (Обычный класс):
  Abstract class (Абстрактный класс);
  Final class (Финализированный класс).
- Interfaces (Интерфейс).

- Enum (Перечисление).
- Nested class (Вложенный класс):
  Static nested class (Статический вложенный класс);
  Member inner class (Простой внутренний класс);
  Local inner class (Локальный класс);
  Anonymous inner class (Анонимный класс).

Расскажите про вложенные классы. В каких случаях они применяются?

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

Существуют четыре категории вложенных классов: + Static nested class (Статический вложенный класс); + Member inner class (Простой внутренний класс); + Local inner class (Локальный класс); + Anonymous inner class (Анонимный класс).

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

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

Что такое «статический класс»?

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

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

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

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

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

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

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

Что такое «локальный класс»? Каковы его особенности?

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

Локальные классы имеют следующие особенности:

- Видны только в пределах блока, в котором объявлены;
- Не могут быть объявлены как private/public/protected или static;
- Не могут иметь внутри себя статических объявлений методов и классов, но могут иметь финальные статические поля, проинициализированные константой;
- Имеют доступ к полям и методам обрамляющего класса;
- Могут обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final.

Что такое «анонимные классы»? Где они применяются?

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

Анонимные классы имеют несколько ограничений:

- Их использование разрешено только в одном месте программы - месте его создания;
- Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться;
- Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.

Анонимные классы обычно применяются для:

- создания объекта функции (function object), например, реализация интерфейса Comparator;
- создания объекта процесса (process object), такого как экземпляры классов Thread, Runnable и подобных;
- в статическом методе генерации;
- инициализации открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.

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

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

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

Для чего используется оператор assert?

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

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

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

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

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

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

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

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

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

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

Различия между Heap и Stack памятью:

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

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

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

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

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

Размер памяти стека намного меньше памяти в куче.

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

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

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

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

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

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

Для чего нужен сборщик мусора?

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

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

Освобождать память от мусора.

Существует два подхода к обнаружению мусора:

- Reference counting;
- Tracing

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

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

Существует 4 типа корневых точки:

- Локальные переменные и параметры методов;
- Потоки;
- Статические переменные;
- Ссылки из JNI.

Самое простое java приложение будет иметь корневые точки:

- Локальные переменные внутри main() метода и параметры main() метода;
- Поток который выполняет main();
- Статические переменные класса, внутри которого находится main() метод.

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

Для очистки памяти от мусора существуют два основных метода:

- Copying collectors
- Mark-and-sweep

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

Объекты создаются в «from-space»;
Когда «from-space» заполняется, приложение приостанавливается;
Запускается сборщик мусора. Находятся живые объекты в «from-space» и копируются в «to-space»;
Когда все объекты скопированы «from-space» полностью очищается;
«to-space» и «from-space» меняются местами.

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

Приложение должно быть остановлено на время, необходимое для полного прохождения цикла сборки мусора;
В худшем случае (когда все объекты живые) «form-space» и «to-space» будут обязаны быть одинакового размера.

Алгоритм работы mark-and-sweep можно описать так:

- Объекты создаются в памяти;
- В момент, когда нужно запустить сборщик мусора приложение приостанавливается;
- Сборщик проходится по дереву объектов, помечая живые объекты;
- Сборщик проходится по всей памяти, находя все не отмеченные куски памяти и сохраняя их в «free list»;
- Когда новые объекты начинают создаваться они создаются в памяти доступной во «free list».

Минусы этого способа:

- Приложение не работает пока происходит сборка мусора;
- Время остановки напрямую зависит от размеров памяти и количества объектов;
- Если не использовать «compacting» память будет использоваться не эффективно.

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

- большинство создаваемых объектов быстро становятся мусором;
- существует мало связей между объектами, которые были созданы в прошлом и только что созданными объектами.

Как работает сборщик мусора?

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

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

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

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

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

Какие разновидности сборщиков мусора реализованы в виртуальной машине HotSpot?

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

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

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

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

опцией -XX:+UseConcMarkSweepGC.

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

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

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

Память приложения делится на три пространства:

- Young generation. Объекты создаются именно в этом участке памяти.
- Old generation. В этот участок памяти перемещаются объекты, которые переживают «minor garbage collection».
- Permanent generation. Тут хранятся метаданные об объектах, Class data sharing (CDS), пул строк (String pool). Permanent область делится на две: только для чтения и для чтения-записи. Очевидно, что в этом случае область только для чтения не чистится сборщиком мусора никогда.

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

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

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

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

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

В Old generation сборка мусора происходит с использованием алгоритма mark-sweep-compact, ко-

торый состоит из трёх фаз. В фазе Mark (пометка) сборщик мусора помечает все живые объекты, затем, в фазе Sweep (очистка) все не помеченные объекты удаляются, а в фазе 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 мегабайт кучи, работающих на машинах клиентского типа и не предъявляющих жёстких требований к величине пауз, затрачиваемых на сборку мусора. В то же время модель «stop-the-world» может вызвать длительные паузы в работе приложения при использовании больших объёмов памяти. Кроме того, последовательный алгоритм работы не позволяет оптимально использовать вычислительные ресурсы компьютера, и последовательный сборщик мусора может стать узким местом при работе приложения на многопроцессорных машинах.

Что такое «пул строк»?

Пул строк – это набор строк, хранящийся в Heap.

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

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

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

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

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

Что такое finalize()? Зачем он нужен?

Через вызов метода finalize() (который наследуется от Java.lang.Object) JVM реализуется функциональность аналогичная функциональности деструкторов в 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.

Пример:

```
public class MainClass {

	public static void main(String args[]) {
		TestClass a = new TestClass();
		a.a();
		a = null;
		a = new TestClass();
		a.a();
		System.out.println(«!!! done»);
	}
}

public class TestClass {

	public void a() {
		System.out.println(«!!! a() called»);
	}

	@Override
	protected void finalize() throws Throwable {
		System.out.println(«!!! finalize() called»);
		super.finalize();
	}
}
```

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

```
!!! a() called
!!! a() called
!!! done
```

Теперь несколько усложним программу, добавив принудительный вызов Garbage Collector:

```
public class MainClass {

	public static void main(String args[]) {
		TestClass a = new TestClass();
		a.a();
		a = null;
		System.gc(); // Принудительно зовём сборщик мусора
		a = new TestClass();
		a.a();
		System.out.println(«!!! done»);
	}

}
```

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

```
!!! a() called
!!! a() called
!!! done
```

!!! finalize() called

Вариант б:

!!! a() called
!!! a() called
!!! finalize() called
!!! done

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

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

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

Чем отличаются final, finally и finalize()?

Модификатор final:

- Класс не может иметь наследников;
- Метод не может быть переопределен в классах наследниках;
- Поле не может изменить свое значение после инициализации;
- Локальные переменные не могут быть изменены после присвоения им значения;
- Параметры методов не могут изменять своё значение внутри метода.

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

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

Пример:

```
public class MainClass {

    public static void main(String args[]) {
        TestClass a = new TestClass();
        System.out.println(«result of a.a() is « + a.a());
        a = null;
        System.gc(); // Принудительно зовём сборщик мусора
        a = new TestClass();
        System.out.println(«result of a.a() is « + a.a());
```

```
            System.out.println(«!!! done»);
        }

}

public class TestClass {

        public int a() {
            try {
                System.out.println(«!!! a() called»);
                throw new Exception(«»);
            } catch (Exception e) {
                System.out.println(«!!! Exception in a()»);
                return 2;
            } finally {
                System.out.println(«!!! finally in a() «);
            }
        }

        @Override
        protected void finalize() throws Throwable {
            System.out.println(«!!! finalize() called»);
            super.finalize();
        }
}
```

Результат выполнения:

```
!!! a() called
!!! Exception in a()
!!! finally in a()
result of a.a() is 2
!!! a() called
!!! Exception in a()
!!! finally in a()
!!! finalize() called
result of a.a() is 2
!!! done
```

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

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

В Java существуют несколько разновидностей приведения:

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

Расширение (повышение, upcasting) примитивного типа (widening primitive). Означает, что осуществляется переход от менее емкого типа к более ёмкому. Например, от типа byte (длина 1 байт) к типу int (длина 4 байта). Такие преобразование безопасны в том смысле, что новый тип всегда

гарантировано вмещает в себя все данные, которые хранились в старом типе и таким образом не происходит потери данных. Этот тип приведения всегда допустим и происходит автоматически.

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

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

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

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

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

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

Для проверки возможности приведения нужно воспользоваться оператором instanceof:

```
Parent parent = new Child();
if (parent instanceof Child) {
   Child child = (Child) parent;
}
```

Когда в приложении может быть выброшено исключение ClassCastException?

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

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

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

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

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

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

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

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

автоупаковку примитива в соответствующий класс-обертку. Однако, в этом случае существуют два дополнительных ограничения: a) присвоение примитива обертке может производится только оператором = (нельзя передать такой примитив в параметры метода без явного приведения типов) b) тип левого операнда не должен быть старше чем Character, тип правого не дожен 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 я вляется то, что они кэшируются JVM. Поэтому такие обертки с одинаковыми значениями будут являться ссылками на один объект.

Какие есть особенности класса String?

Это неизменяемый (immutable) и финализированный тип данных;
Все объекты класса String JVM хранит в пуле строк;
Объект класса String можно получить, используя двойные кавычки;
Можно использовать оператор + для конкатенации строк;
Начиная с Java 7 строки можно использовать в конструкции switch.

Почему String неизменяемый и финализированный класс?

Есть несколько преимуществ в неизменности строк:

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

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

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

Строки используются classloader и неизменность обеспечивает правильность загрузки класса.

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

Почему char[] предпочтительнее String

для хранения пароля?

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

Почему строка является популярным ключом в HashMap в Java?

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

Что делает метод intern() в классе String?

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

Можно ли использовать строки в конструкции switch?

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

участвующие строки чувствительны к регистру;
используется метод equals() для сравнения полученного значения со значениями case, поэтому во избежание NullPointerException стоит предусмотреть проверку на null.
согласно документации, Java 7 для строк в switch, компилятор Java формирует более эффективный байткод для строк в конструкции switch, чем для сцепленных условий if-else.

Какая основная разница между String, StringBuffer, StringBuilder?

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

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

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

Что такое класс Object? Какие в нем есть методы?

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

public boolean equals(Object obj) – служит для сравнения объектов по значению; int hashCode() – возвращает hash код для объекта; String toString() – возвращает строковое представление объекта; Class getClass() – возвращает класс объекта во время выполнения; protected Object clone() – соз-

дает и возвращает копию объекта; void notify() – возобновляет поток, ожидающий монитор; void notifyAll() – возобновляет все потоки, ожидающие монитор; void wait() – остановка вызвавшего метод потока до момента пока другой поток не вызовет метод notify() или notifyAll() для этого объекта; void wait(long timeout) – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод notify() или notifyAll() для этого объекта; void wait(long timeout, int nanos) – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод notify() или notifyAll() для этого объекта; protected void finalize() – может вызываться сборщиком мусора в момент удаления объекта при сборке мусора.

Дайте определение понятию «конструктор».

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

Что такое «конструктор по умолчанию»?

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

```
public class ClassName() {}
```

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

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

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

Где и как вы можете использовать приватный конструктор?

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

Расскажите про классы-загрузчики и про динамическую загрузку классов.

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

В начале работы программы создается 3 основных загрузчика классов:

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

гаются в директории <JAVA_HOME>/lib/ext или другой директории, описанной в системном параметре java.ext.dirs. Это позволяет обновлять и добавлять новые расширения без необходимости модифицировать настройки используемых приложений. Загрузчик расширений реализован классом sun.misc.Launcher$ExtClassLoader.

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

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

Процесс загрузки класса состоит из трех частей:

- Loading – на этой фазе происходит поиск и физическая загрузка файла класса в определенном источнике (в зависимости от загрузчика). Этот процесс определяет базовое представление класса в памяти. На этом этапе такие понятия как «методы», «поля» и т.д. пока не известны.
- Linking – процесс, который может быть разбит на 3 части:
  Bytecode verification – проверка байт-кода на соответствие требованиям, определенным в спецификации JVM.
  Class preparation – создание и инициализация необходимых структур, используемых для представления полей, методов, реализованных интерфейсов и т.п., определенных в загружаемом классе.
  Resolving – загрузка набора классов, на которые ссылается загружаемый класс.
- Initialization – вызов статических блоков инициализации и присваивание полям класса значений по умолчанию.

Динамическая загрузка классов в Java имеет ряд особенностей:

- отложенная (lazy) загрузка и связывание классов. Загрузка классов производится только при необходимости, что позволяет экономить ресурсы и распределять нагрузку.
- проверка корректности загружаемого кода (type safeness). Все действия связанные с контролем использования типов производятся только во время загрузки класса, позволяя избежать дополнительной нагрузки во время выполнения кода.
- программируемая загрузка. Пользовательский загрузчик полностью контролирует процесс получения запрошенного класса — самому ли искать байт-код и создавать класс или делегировать создание другому загрузчику. Дополнительно существует возможность выставлять различные атрибуты безопасности для загружаемых классов, позволяя таким образом работать с кодом из ненадежных источников.
- множественные пространства имен. Каждый загрузчик имеет своё пространство имён для создаваемых классов. Соответственно, классы, загруженные двумя различными загрузчиками на основе общего байт-кода, в системе будут различаться.

Существует несколько способов инициировать загрузку требуемого класса:
- явный: вызов ClassLoader.loadClass() или Class.forName() (по умолчанию используется загрузчик, создавший текущий класс, но есть возможность и явного указания загрузчика);
- неявный: когда для дальнейшей работы приложения требуется ранее не использованный класс, JVM инициирует его загрузку.

Что такое Reflection?

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

Возможности Java Reflection API:

Определение класса объекта;
Получение информации о модификаторах класса, полях, методах, конструкторах и суперклассах;
Определение интерфейсов, реализуемых классом;
Создание экземпляра класса;
Получение и установка значений полей объекта;
Вызов методов объекта;
Создание нового массива.

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

Метод equals() - определяет отношение эквивалентности объектов.

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

Если вы хотите переопределить equals(), какие условия должны выполняться? Какими свойствами обладает порождаемое equals() отношение эквивалентности?

Рефлексивность: для любой ссылки на значение x, x.equals(x) вернет true;
Симметричность: для любых ссылок на значения x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) возвращает true.
Транзитивность: для любых ссылок на значения x, y и z, если x.equals(y) и y.equals(z) возвращают true, тогда и x.equals(z) вернёт true;
Непротиворечивость: для любых ссылок на значения 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.

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

Правила переопределения метода Object.equals().

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

Какая связь между hashCode() и equals()? Если equals() переопределен, есть ли какие-либо другие методы, которые следует переопределить?

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

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

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

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

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

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

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

```
public native int hashCode();
```

Для чего нужен метод hashCode()?

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

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

Каковы правила переопределения метода Object.hashCode()? Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode()?

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

Могут ли у разных объектов быть одинаковые hashCode()?

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

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

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

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

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

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

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

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

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

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

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

В equals() требуется проверять, что аргумент equals(Object that) такого же типа что и сам объект.

В чем разница между this.getClass() == that.getClass() и that instanceof MyClass?

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

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

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

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

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

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

Расскажите про клонирование объектов.

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

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

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

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

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс Cloneable и переопределяет метод clone(). Так как, если это будет иначе вызов метода невозмо-

жен из-за его недоступности. Отсюда следует, что если класс имеет суперкласс, то для реализации механизма клонирования текущего класса-потомка необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений final для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

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

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

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

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

В чем отличие между поверхностным и глубоким клонированием?

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

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

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

Глубокое копирование дублирует абсолютно всю информацию объекта:

Нет необходимости копировать отдельно примитивные данные;

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

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

Какой способ клонирования предпочтительней?

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

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

Поля для клонирования указываются явно;

Возможность клонировать даже final поля.

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

Метод clone() объявлен в классе Object с указанием модификатора native, чтобы обеспечить доступ

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

Опишите иерархию исключений.

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

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

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

Какие виды исключений в Java вы знаете, чем они отличаются? Что такое checked и unchecked exception?

В Java все исключения делятся на два типа:

- checked (контролируемые/проверяемые исключения) должны обрабатываться блоком catch или описываться в 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.

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.

Как написать собственное («пользовательское») исключение?

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

```
class CustomException extends Exception {
  public CustomException() {
    super();
  }

  public CustomException(final String string) {
    super(string + « is invalid»);
  }

  public CustomException(final Throwable cause) {
    super(cause);
  }
}
```

Какие существуют unchecked exception?

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

Что представляет из себя ошибки класса Error?

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

Что вы знаете о OutOfMemoryError?

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

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

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

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

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

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

Опишите работу блока try-catch-finally.

try — данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке. catch — ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений в случае их возникновения. finally — ключевое слово для

отметки начала блока кода, который является дополнительным. Этот блок помещается после последнего блока catch. Управление передаётся в блок finally в любом случае, было выброшено исключение или нет.

Общий вид конструкции для обработки исключительной ситуации выглядит следующим образом:

```
try {
   //код, который потенциально может привести к исключительной ситуации
}
catch(SomeException e ) { //в скобках указывается класс конкретной ожидаемой ошибки
   //код обработки исключительной ситуации
}
finally {
   //необязательный блок, код которого выполняется в любом случае
}
```

Что такое механизм try-with-resources?

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

Общий вид конструкции:

```
try(/*объявление ресурсов*/) {
   //...
} catch(Exception ex) {
   //...
} finally {
   //...
}
```

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

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

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

Может ли один блок catch отлавливать сразу несколько исключений?

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

```
try {
   //...
} catch(IOException | SQLException ex) {
   //...
}
```

Всегда ли исполняется блок finally?

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

Существуют ли ситуации, когда блок finally не будет выполнен?

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

```
try {
   System.exit(0);
} catch(Exception e) {
   e.printStackTrace();
} finally { }
```

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

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

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

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

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

```
void method() {
   try {
      //...
   } catch (FileNotFoundException ex) {
      //...
   } catch (IOException ex) {
      //...
   }
}
```

Что такое generics?

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

Примером использования обобщенных типов может служить Java Collection Framework. Так, класс LinkedList<E> - типичный обобщенный тип. Он содержит параметр E, который представляет тип

элементов, которые будут храниться в коллекции. Создание объектов обобщенных типов происходит посредством замены параметризированных типов реальными типами данных. Вместо того, чтобы просто использовать LinkedList, ничего не говоря о типе элемента в списке, предлагается использовать точное указание типа LinkedList<String>, LinkedList<Integer> и т.п.

Что такое «интернационализация», «локализация»?

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

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

JAVA Collections

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 предопределённым способом вставки и извлечения FIFO (first-in-first-out):
PriorityQueue — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».
ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out).

Интерфейс Map реализован классами:

Hashtable — хэш-таблица, методы которой синхронизированы. Не позволяет использовать null в качестве значения или ключа и не является упорядоченной.
HashMap — хэш-таблица. Позволяет использовать null в качестве значения или ключа и не является упорядоченной.
LinkedHashMap — упорядоченная реализация хэш-таблицы.
TreeMap — реализация, основанная на красно-чёрных деревьях. Является упорядоченной и предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».
WeakHashMap — реализация хэш-таблицы, которая организована с использованием weak references для ключей (сборщик мусора автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элемента нет жёстких ссылок).

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

Iterable
  Collection
    List
    Set
      SortedSet
        NavigableSet
Map
  SortedMap
    NavigableMap
Iterator

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

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

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

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

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

Что такое «fail-fast поведение»?

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

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

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

- при изменении коллекции счетчик модификаций так же изменяется;
- при создании итератора ему передается текущее значение счетчика;
- при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение.

Какая разница между fail-fast и fail-safe?

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

Приведите примеры итераторов, реализующих поведение fail-safe

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

Чем различаются Enumeration и Iterator.

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

- с помощью Enumeration нельзя добавлять/удалять элементы;
- в Iterator исправлены имена методов для повышения читаемости кода (Enumeration. hasMoreElements() соответствует Iterator.hasNext(), Enumeration.nextElement() соответствует Iterator. next() и т.д);
- Enumeration присутствуют в устаревших классах, таких как Vector/Stack, тогда как Iterator есть во всех современных классах-коллекциях.

Как между собой связаны Iterable и Iterator?

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

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

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

Сравните Iterator и ListIterator.

ListIterator расширяет интерфейс Iterator

ListIterator может быть использован только для перебора элементов коллекции List;

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

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

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

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

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

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

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

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

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

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

При следующем вызове методов итератора будет выброшено ConcurrentModificationException.

Как избежать ConcurrentModificationException во время перебора коллекции?

- Попробовать подобрать 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.

Какая коллекция реализует дисциплину обслуживания FIFO?

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

Какая коллекция реализует дисциплину обслуживания FILO?

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

Чем отличается ArrayList от Vector? Зачем добавили ArrayList, если уже был Vector?

Методы класса Vector синхронизированы, а ArrayList - нет;

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

Vector это устаревший класс и его использование не рекомендовано.

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

ArrayList это список, реализованный на основе массива, а LinkedList — это классический двусвяз-

ный 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 константное O(1);
- 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 с списке.

Какое худшее время работы метода add() дляLinkedList?

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. Но если говорить об общей эффективности - то самый быстрый способ будет с использованием System.arraycopy(), и получить к нему доступ можно через метод - subList(int fromIndex, int toIndex)

Пример:

```
import java.io.*;
import java.util.ArrayList;

public class Main {
   //позиция с которой удаляем
   private static int m = 0;
   //количество удаляемых элементов
   private static int n = 0;
   //количество элементов в списке
   private static final int size = 1000000;
   //основной список (для удаления вызовом remove() и его копия для удаления путём перезаписи)
   private static ArrayList<Integer> initList, copyList;

   public static void main(String[] args){

      initList = new ArrayList<>(size);
      for (int i = 0; i < size; i++) {
         initList.add(i);
      }
      System.out.println(«Список из 1.000.000 элементов заполнен»);
```

```
    copyList = new ArrayList<>(initList);
    System.out.println(«Создана копия списка\n»);

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    try{
      System.out.print(«С какой позиции удаляем? > «);
      m = Integer.parseInt(br.readLine());
      System.out.print(«Сколько удаляем? > «);
      n = Integer.parseInt(br.readLine());
    } catch(IOException e){
      System.err.println(e.toString());
    }
    System.out.println(«\nВыполняем удаление вызовом remove()...»);
    long start = System.currentTimeMillis();

    for (int i = m - 1; i < m + n - 1; i++) {
      initList.remove(i);
    }

    long finish = System.currentTimeMillis() - start;
    System.out.println(«Время удаления с помощью вызова remove(): « + finish);
    System.out.println(«Размер исходного списка после удаления: « + initList.size());

    System.out.println(«\nВыполняем удаление путем перезаписи...\n»);
    start = System.currentTimeMillis();

    removeEfficiently();

    finish = System.currentTimeMillis() - start;
    System.out.println(«Время удаления путём смещения: « + finish);
    System.out.println(«Размер копии списка:» + copyList.size());

    System.out.println(«\nВыполняем удаление через SubList...\n»);
    start = System.currentTimeMillis();

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

    finish = System.currentTimeMillis() - start;
    System.out.println(«Время удаления через саблист: « + finish);
    System.out.println(«Размер копии списка:» + copyList.size());
}

private static void removeEfficiently(){
    /* если необходимо удалить все элементы, начиная с указанного,
     * то удаляем элементы с конца до m
     */
    if (m + n >= size){
      int i = size - 1;
      while (i != m - 1){
        copyList.remove(i);
        i--;
      }
    } else{
      //переменная k необходима для отсчёта сдвига начиная от места вставка m
```

```
        for (int i  = m + n, k = 0; i < size; i++, k++) {
          copyList.set(m + k, copyList.get(i));
        }

        /* удаляем ненужные элементы в конце списка
         * удаляется всегда последний элемент, так как время этого действия
         * фиксировано и не зависит от размера списка
         */
        int i = size - 1;
        while (i != size - n - 1){
           copyList.remove(i);
           i--;
        }
        //сокращаем длину списка путём удаления пустых ячеек
        copyList.trimToSize();
      }
    }
}
```

Результат выполнения:

```
run:
Список из 1.000.000 элементов заполнен
Создана копия списка

С какой позиции удаляем? > 600000
Сколько удаляем? > 20000

Выполняем удаление вызовом remove()...
Время удаления с помощью вызова remove(): 928
Размер исходного списка после удаления: 980000

Выполняем удаление путем перезаписи...

Время удаления путём смещения: 17
Размер копии списка:980000

Выполняем удаление через SubList...

Время удаления через саблист: 1
Размер копии списка:980000
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 33 секунды)
```

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

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

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

Создается один новый экземпляр вложенного класса Node.

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

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

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
//...
}
```

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

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

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

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

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.

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.

Почему LinkedList реализует и List, и Deque?

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

LinkedList — это односвязный, двусвязный или четырехсвязный список?

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

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

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

Что позволяет сделать PriorityQueue?

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

коллекция не поддерживает null в качестве элементов.

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

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

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

Зачем нужен HashMap, если есть Hashtable?

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

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

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

Hashtable это устаревший класс и его использование не рекомендовано.

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

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

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

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

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

Пример кода:

```
import java.util.HashMap;
import java.util.IdentityHashMap;
```

```
import java.util.Map;

public class Q2 {

   public static void main(String[] args) {
      Q2 q = new Q2();
      q.testHashMapAndIdentityHashMap();
   }

   private void testHashMapAndIdentityHashMap() {
      CreditCard visa = new CreditCard(«VISA», «04/12/2019»);

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

      System.out.println(«adding to HM»);
      // inserting objects to HashMap
      cardToExpiry.put(visa, visa.getExpiryDate());

      // inserting objects to IdentityHashMap
      cardToExpiryIdenity.put(visa, visa.getExpiryDate());
      System.out.println(«adding to IHM»);

      System.out.println(«before modifying keys»);
      String result = cardToExpiry.get(visa) != null ? «Yes» : «No»;
      System.out.println(«Does VISA card exists in HashMap? « + result);

      result = cardToExpiryIdenity.get(visa) != null ? «Yes» : «No»;
      System.out.println(«Does VISA card exists in IdenityHashMap? « + result);

      // modifying value object
      visa.setExpiryDate(«02/11/2030»);

      System.out.println(«after modifying keys»);
      result = cardToExpiry.get(visa) != null ? «Yes» : «No»;
      System.out.println(«Does VISA card exists in HashMap? « + result);

      result = cardToExpiryIdenity.get(visa) != null ? «Yes» : «No»;
      System.out.println(«Does VISA card exists in IdenityHashMap? « + result);

      System.out.println(«cardToExpiry.containsKey»);
      System.out.println(cardToExpiry.containsKey(visa));
      System.out.println(«cardToExpiryIdenity.containsKey»);
      System.out.println(cardToExpiryIdenity.containsKey(visa));
   }

}

class CreditCard {
   private String issuer;
   private String expiryDate;

   public CreditCard(String issuer, String expiryDate) {
      this.issuer = issuer;
```

```
        this.expiryDate = expiryDate;
    }

    public String getIssuer() {
        return issuer;
    }

    public String getExpiryDate() {
        return expiryDate;
    }

    public void setExpiryDate(String expiry) {
        this.expiryDate = expiry;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((expiryDate == null) ? 0 : expiryDate.hashCode());
        result = prime * result + ((issuer == null) ? 0 : issuer.hashCode());
        System.out.println(«hashCode = « + result);
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        System.out.println(«equals !!! «);
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        CreditCard other = (CreditCard) obj;
        if (expiryDate == null) {
            if (other.expiryDate != null)
                return false;
        } else if (!expiryDate.equals(other.expiryDate))
            return false;
        if (issuer == null) {
            if (other.issuer != null)
                return false;
        } else if (!issuer.equals(other.issuer))
            return false;
        return true;
    }

}
```

Результат выполнения кода:

```
adding to HM
hashCode = 1285631513
```

adding to IHM
before modifying keys
hashCode = 1285631513
Does VISA card exists in HashMap? Yes
Does VISA card exists in IdenityHashMap? Yes
after modifying keys
hashCode = 791156485
Does VISA card exists in HashMap? No
Does VISA card exists in IdenityHashMap? Yes
cardToExpiry.containsKey
hashCode = 791156485
false
cardToExpiryIdenity.containsKey
true

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

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

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

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

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

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

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

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

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

Реализация LinkedHashMap отличается от HashMap поддержкой двухсвязанного списка, определяющего порядок итерации по элементам структуры данных. По умолчанию элементы списка упоря-

дочены согласно их порядку добавления в LinkedHashMap (insertion-order). Однако порядок итерации можно изменить, установив параметр конструктора accessOrder в значение true. В этом случае доступ осуществляется по порядку последнего обращения к элементу (access-order). Это означает, что при вызове методов get() или put() элемент, к которому обращаемся, перемещается в конец списка.

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

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

Так же оно проявляется при итерации по коллекции.

Как устроен HashMap?

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

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

HashMap реализован с использованием метода цепочек, т.е. каждой ячейке массива (корзине) соответствует свой связный список и при возникновении коллизии осуществляется добавление нового элемента в этот список.
Для метода цепочек коэффициент заполнения может быть больше 1 и с увеличением числа элементов производительность убывает линейно. Такие таблицы удобно использовать, если заранее неизвестно количество хранимых элементов, либо их может быть достаточно много, что приводит к большим значениям коэффициента заполнения.

Среди методов открытой реализации различают:

- линейное пробирование;
- квадратичное пробирование;
- двойное хэширование.

Недостатки структур с методом открытой адресации:

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

Сложно организовать удаление элемента.

Первые два метода открытой адресации приводят к проблеме первичной и вторичной группиро-

вок.

Преимущества хэш-таблицы с открытой адресацией:

- отсутствие затрат на создание и хранение объектов списка;
- простота организации сериализации/десериализации объекта.

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

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

Какое начальное количество корзин в HashMap?

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

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

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

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

Пример кода двоичного поиска:

```
public class Q {
  public static void main(String[] args) {
    Q q = new Q();
    q.binSearch();
  }

  private void binSearch() {
    int[] inpArr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    Integer result = binSearchF(inpArr, 1, 0, inpArr.length - 1);
    System.out.println(«----------»);
    result = binSearchF(inpArr, 2, 0, inpArr.length - 1);
    System.out.println(«Found at position « + result);
  }

  private Integer binSearchF(int[] inpArr, int searchValue, int low, int high) {
    Integer index = null;
    while (low <= high) {
      System.out.println(«New iteration, low = « + low + «, high = « + high);
      int mid = (low + high) / 2;
```

```
        System.out.println(«trying mid = « + mid + « inpArr[mid] = « + inpArr[mid]);
        if (inpArr[mid] < searchValue) {
          low = mid + 1;
          System.out.println(«inpArr[mid] (« + inpArr[mid] + «) < searchValue(« + searchValue + «),
mid = « + mid
              + «, setting low = « + low);
        } else if (inpArr[mid] > searchValue) {
          high = mid - 1;
          System.out.println(«inpArr[mid] (« + inpArr[mid] + «) > searchValue(« + searchValue + «),
mid = « + mid
              + «, setting high = « + high);
        } else if (inpArr[mid] == searchValue) {
          index = mid;
          System.out.println(«found at index « + mid);
          break;
        }
      }
      return index;
    }
}
```

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

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

В каком случае может быть потерян элемент в HashMap?

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

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

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

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

hashCode позволяет определить корзину для поиска элемента, а equals используется для сравнения

ключей элементов в списке корзины и искомого ключа.

Каково максимальное число значений hashCode()?

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

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

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

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

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

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

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

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

Как и когда происходит увеличение количества корзин в HashMap?

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

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

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

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

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

Да, будет, но в этом случае HashMap вырождается в связный список и теряет свои преимущества.
Как перебрать все ключи Map?

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

Как перебрать все значения Map?

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

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

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

В чем отличия TreeSet и HashSet?

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

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

Что будет, если добавлять элементы в TreeSet по возрастанию?

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

Чем LinkedHashSet отличается от HashSet?

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

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

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

Все основные операции выполняются за O(1) и обычно (но негарантированно) быстрей аналогов из HashSet, а пакетные операции (bulk operations), такие как containsAll() и retainAll() выполняются даже горазда быстрей.
Помимо всего EnumSet предоставляет множество статических методов инициализации для упрощенного и удобного создания экземпляров.

Какие существуют способы перебирать элементы списка?

Цикл с итератором

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
  //iterator.next();
}
```

Цикл for

```
for (int i = 0; i < list.size(); i++) {
  //list.get(i);
}
```

Цикл while

```
int i = 0;
while (i < list.size()) {
  //list.get(i);
  i++;
}
```

«for-each»

```
for (String element : list) {
  //element;
}
```

Каким образом можно получить синхронизированные объекты стандартных коллекций?

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

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
```

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

Как получить коллекцию только для чтения?

При помощи:

```
Collections.unmodifiableList(list);
Collections.unmodifiableSet(set);
Collections.unmodifiableMap(map).
```

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

Напишите однопоточную программу, которая заставляет коллекцию выбросить ConcurrentModificationException.

```
public static void main(String[] args) {
   List<Integer> list = new ArrayList<>();
   list.add(1);
   list.add(2);
   list.add(3);

   for (Integer integer : list) {
      list.remove(1);
   }
}
```

Приведите пример, когда какая-либо коллекция выбрасывает UnsupportedOperationException.

```
public static void main(String[] args) {
   List<Integer> list = Collections.emptyList();
   list.add(0);
}
```

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

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

```
<T> Collection<T> symmetricDifference(Collection<T> a, Collection<T> b) {
   // Объединяем коллекции.
   Collection<T> result = new ArrayList<>(a);
   result.addAll(b);

   // Получаем пересечение коллекций.
   Collection<T> intersection = new ArrayList<>(a);
   intersection.retainAll(b);

   // Удаляем элементы, расположенные в обоих коллекциях.
   result.removeAll(intersection);

   return result;
}
```

Как, используя LinkedHashMap, сделать кэш с «invalidation policy»?

Необходимо использовать LRU-алгоритм (Least Recently Used algorithm) и LinkedHashMap с access-

order. В этом случае при обращении к элементу он будет перемещаться в конец списка, а наименее используемые элементы будут постепенно группироваться в начале списка. Так же в стандартной реализации LinkedHashMap есть метод removeEldestEntries(), который возвращает true, если текущий объект LinkedHashMap должен удалить наименее используемый элемент из коллекции при использовании методов put() и putAll().

```
public class LRUCache<K, V> extends LinkedHashMap<K, V> {
   private static final int MAX_ENTRIES = 10;

   public LRUCache(int initialCapacity) {
      super(initialCapacity, 0.85f, true);
   }

   @Override
   protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
      return size() > MAX_ENTRIES;
   }
}
```

Стоит заметить, что LinkedHashMap не позволяет полностью реализовать LRU-алгоритм, поскольку при вставке уже имеющегося в коллекции элемента порядок итерации по элементам не меняется.

Как одной строчкой скопировать элементы любой collection в массив?

```
Object[] array = collection.toArray();
```

Как одним вызовом из List получить List со всеми элементами, кроме первых и последних 3-х?

```
List<Integer> subList = list.subList(3, list.size() - 3);
```

Как одной строчкой преобразовать HashSet в ArrayList?

```
ArrayList<Integer> list = new ArrayList<>(new HashSet<>());
```

Как одной строчкой преобразовать ArrayList в HashSet?

```
HashSet<Integer> set = new HashSet<>(new ArrayList<>());
```

Сделайте HashSet из ключей HashMap.

```
HashSet<Object> set = new HashSet<>(map.keySet());
```

Сделайте HashMap из HashSet <Map.Entry<K, V>>.

```
HashMap<K, V> map = new HashMap<>(set.size());
for (Map.Entry<K, V> entry : set) {
   map.put(entry.getKey(), entry.getValue());
}
```

JAVA 8

Какие нововведения, появились в Java 8 и JDK 8?

Методы интерфейсов по умолчанию;
Лямбда-выражения;
Функциональные интерфейсы;
Ссылки на методы и конструкторы;
Повторяемые аннотации;
Аннотации на типы данных;
Рефлексия для параметров методов;
Stream API для работы с коллекциями;
Параллельная сортировка массивов;
Новое API для работы с датами и временем;
Новый движок JavaScript Nashorn;
Добавлено несколько новых классов для потокобезопасной работы;
Добавлен новый API для Calendar и Locale;
Добавлена поддержка Unicode 6.2.0;
Добавлен стандартный класс для работы с Base64;
Добавлена поддержка беззнаковой арифметики;
Улучшена производительность конструктора java.lang.String(byte[], *) и метода java.lang.String.getBytes();
Новая реализация AccessController.doPrivileged, позволяющая устанавливать подмножество привилегий без необходимости проверки всех остальных уровней доступа;
Password-based алгоритмы стали более устойчивыми;
Добавлена поддержка SSL/TLS Server Name Indication (NSI) в JSSE Server;
Улучшено хранилище ключей (KeyStore);
Добавлен алгоритм SHA-224;
Удален мост JDBC - ODBC;
Удален PermGen, изменен способ хранения мета-данных классов;
Возможность создания профилей для платформы Java SE, которые включают в себя не всю платформу целиком, а некоторую ее часть;
Инструментарий
- Добавлена утилита jjs для использования JavaScript Nashorn;
- Команда java может запускать JavaFX приложения;
- Добавлена утилита jdeps для анализа .class-файлов.

Что такое «лямбда»? Какова структура и особенности использования лямбда-выражения?

Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

```
interface Operationable {
   int calculate(int x, int y);
}

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

```
    Operationable operation = (x, y) -> x + y;
    int result = operation.calculate(10, 20);
    System.out.println(result); //30
}
```

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java.

Отложенное выполнение (deferred execution) лямбда-выражения- определяется один раз в одном месте программы, вызываются при необходимости, любое количество раз и в произвольном месте программы.

Параметры лямбда-выражения должны соответствовать по типу параметрам метода функционального интерфейса:

```
operation = (int x, int y) -> x + y;
//При написании самого лямбда-выражения тип параметров разрешается не указывать:
(x, y) -> x + y;
//Если метод не принимает никаких параметров, то пишутся пустые скобки, например,
() -> 30 + 20;
//Если метод принимает только один параметр, то скобки можно опустить:
n -> n * n;
```

Конечные лямбда-выражения не обязаны возвращать какое-либо значение.

```
interface Printable {
    void print(String s);
}

public static void main(String[] args) {
    Printable printer = s -> System.out.println(s);
    printer.print("Hello, world");
}
```

Блочные лямбда-выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return:

```
Operationable operation = (int x, int y) -> {
    if (y == 0) {
        return 0;
    }
    else {
        return x / y;
    }
};
```

Передача лямбда-выражения в качестве параметра метода:

```
interface Condition {
    boolean isAppropriate(int n);
}
```

```
private static int sum(int[] numbers, Condition condition) {
   int result = 0;
   for (int i : numbers) {
      if (condition.isAppropriate(i)) {
         result += i;
      }
   }
   return result;
}

public static void main(String[] args) {
   System.out.println(sum(new int[] {0, 1, 0, 3, 0, 5, 0, 7, 0, 9}, (n) -> n != 0));
}
```

К каким переменным есть доступ у лямбда-выражений?

Доступ к переменным внешней области действия из лямбда-выражения очень схож к доступу из анонимных объектов. Можно ссылаться на:

- неизменяемые (effectively final - не обязательно помеченные как final) локальные переменные;
- поля класса;
- статические переменные.

К методам по умолчанию реализуемого функционального интерфейса обращаться внутри лямбда-выражения запрещено.

Как отсортировать список строк с помощью лямбда-выражения?

```
public static List<String> sort(List<String> list){
   Collections.sort(list, (a, b) -> a.compareTo(b));
   return list;
}
```

Что такое «ссылка на метод»?

Если существующий в классе метод уже делает все, что необходимо, то можно воспользоваться механизмом method reference (ссылка на метод) для непосредственной передачи этого метода. Такая ссылка передается в виде:

имя_класса::имя_статического_метода для статического метода;
объект_класса::имя_метода для метода экземпляра;
название_класса::new для конструктора.

Результат будет в точности таким же, как в случае определения лямбда-выражения, которое вызывает этот метод.

```
private interface Measurable {
   public int length(String string);
}

public static void main(String[] args) {
   Measurable a = String::length;
   System.out.println(a.length("abc"));}
```

Ссылки на методы потенциально более эффективны, чем использование лямбда-выражений. Кроме того, они предоставляют компилятору более качественную информацию о типе и при возможности выбора между использованием ссылки на существующий метод и использованием лямбда-выражения, следует всегда предпочитать использование ссылки на метод.

Какие виды ссылок на методы вы знаете?

- на статический метод;
- на метод экземпляра;
- на конструкторе.

Объясните выражение System.out::println.

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

Что такое «функциональные интерфейсы»?

Функциональный интерфейс - это интерфейс, который определяет только один абстрактный метод.

Чтобы точно определить интерфейс как функциональный, добавлена аннотация @ FunctionalInterface, работающая по принципу @Override. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

Интерфейс может включать сколько угодно default методов и при этом оставаться функциональным, потому что default методы - не абстрактные.

Для чего нужны функциональные интерфейсы Function<T,R>, DoubleFunction<R>, IntFunction<R> и LongFunction<R>?

Function<T, R> - интерфейс, с помощью которого реализуется функция, получающая на вход экземпляр класса T и возвращающая на выходе экземпляр класса R.

Методы по умолчанию могут использоваться для построения цепочек вызовов (compose, andThen).

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);
backToString.apply("123");     // "123"
```

DoubleFunction<R> - функция, получающая на вход Double и возвращающая на выходе экземпляр класса R;
IntFunction<R> - функция, получающая на вход Integer и возвращающая на выходе экземпляр класса R;
LongFunction<R> - функция, получающая на вход Long и возвращающая на выходе экземпляр класса R.

Для чего нужны функциональные интерфейсы UnaryOperator<T>, DoubleUnaryOperator, IntUnaryOperator и LongUnaryOperator?

UnaryOperator<T> (унарный оператор) принимает в качестве параметра объект типа T, выполняет

над ними операции и возвращает результат операций в виде объекта типа T:

```
UnaryOperator<Integer> operator = x -> x * x;
System.out.println(operator.apply(5)); // 25
```

DoubleUnaryOperator - унарный оператор, получающий на вход Double;
IntUnaryOperator - унарный оператор, получающий на вход Integer;
LongUnaryOperator - унарный оператор, получающий на вход Long.

Для чего нужны функциональные интерфейсы BinaryOperator<T>, DoubleBinaryOperator, IntBinaryOperator и LongBinaryOperator?

BinaryOperator<T> (бинарный оператор) - интерфейс, с помощью которого реализуется функция, получающая на вход два экземпляра класса T и возвращающая на выходе экземпляр класса T.

```
BinaryOperator<Integer> operator = (a, b) -> a + b;
System.out.println(operator.apply(1, 2)); // 3
```

DoubleBinaryOperator - бинарный оператор, получающий на вход Double;
IntBinaryOperator - бинарный оператор, получающий на вход Integer;
LongBinaryOperator - бинарный оператор, получающий на вход Long.

Для чего нужны функциональные интерфейсы Predicate<T>, DoublePredicate, IntPredicate и LongPredicate?

Predicate<T> (предикат) - интерфейс, с помощью которого реализуется функция, получающая на вход экземпляр класса T и возвращающая на выходе значение типа boolean.

Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия (and, or, negate).

```
Predicate<String> predicate = (s) -> s.length() > 0;
predicate.test("foo"); // true
predicate.negate().test("foo"); // false
```

DoublePredicate - предикат, получающий на вход Double;
IntPredicate - предикат, получающий на вход Integer;
LongPredicate - предикат, получающий на вход Long.

Для чего нужны функциональные интерфейсы Consumer<T>, DoubleConsumer, IntConsumer и LongConsumer?

Consumer<T> (потребитель) - интерфейс, с помощью которого реализуется функция, которая получает на вход экземпляр класса T, производит с ним некоторое действие и ничего не возвращает.

```
Consumer<String> hello = (name) -> System.out.println("Hello, " + name);
hello.accept("world");
```

DoubleConsumer - потребитель, получающий на вход Double;

IntConsumer - потребитель, получающий на вход Integer;
LongConsumer - потребитель, получающий на вход Long.

Для чего нужны функциональные интерфейсы Supplier<T>, BooleanSupplier, DoubleSupplier, IntSupplier и LongSupplier?

Supplier<T> (поставщик) - интерфейс, с помощью которого реализуется функция, ничего не принимающая на вход, но возвращающая на выход результат класса T;

```
Supplier<LocalDateTime> now = LocalDateTime::now;
now.get();
```

DoubleSupplier - поставщик, возвращающий Double;
IntSupplier - поставщик, возвращающий Integer;
LongSupplier - поставщик, возвращающий Long.

Для чего нужен функциональный интерфейс BiConsumer<T,U>?

BiConsumer<T,U> представляет собой операцию, которая принимает два аргумента классов T и U производит с ними некоторое действие и ничего не возвращает.

Для чего нужен функциональный интерфейс BiFunction<T,U,R>?

BiFunction<T,U,R> представляет собой операцию, которая принимает два аргумента классов T и U и возвращающая результат класса R.

Для чего нужен функциональный интерфейс BiPredicate<T,U>?

BiPredicate<T,U> представляет собой операцию, которая принимает два аргумента классов T и U и возвращающая результат типа boolean.

Для чего нужны функциональные интерфейсы вида _To_Function?

DoubleToIntFunction - операция принимающая аргумент класса Double и возвращающая результат типа Integer;
DoubleToLongFunction - операция принимающая аргумент класса Double и возвращающая результат типа Long;
IntToDoubleFunction - операция принимающая аргумент класса Integer и возвращающая результат типа Double;
IntToLongFunction - операция принимающая аргумент класса Integer и возвращающая результат типа Long;
LongToDoubleFunction - операция принимающая аргумент класса Long и возвращающая результат типа Double;
LongToIntFunction - операция принимающая аргумент класса Long и возвращающая результат типа Integer.

Для чего нужны функциональные интерфейсы ToDoubleBiFunction<T,U>, ToIntBiFunction<T,U> и ToLongBiFunction<T,U>?

ToDoubleBiFunction<T,U> - операция принимающая два аргумента классов T и U и возвращающая результат типа Double;

ToLongBiFunction<T,U> - операция принимающая два аргумента классов T и U и возвращающая результат типа Long;

ToIntBiFunction<T,U> - операция принимающая два аргумента классов T и U и возвращающая результат типа Integer.

Для чего нужны функциональные интерфейсы ToDoubleFunction<T>, ToIntFunction<T> и ToLongFunction<T>?

ToDoubleFunction<T> - операция принимающая аргумент класса T и возвращающая результат типа Double;

ToLongFunction<T> - операция принимающая аргумент класса T и возвращающая результат типа Long;

ToIntFunction<T> - операция принимающая аргумент класса T и возвращающая результат типа Integer.

Для чего нужны функциональные интерфейсы ObjDoubleConsumer<T>, ObjIntConsumer<T> и ObjLongConsumer<T>?

ObjDoubleConsumer<T> - операция, которая принимает два аргумента классов T и Double, производит с ними некоторое действие и ничего не возвращает;

ObjLongConsumer<T> - операция, которая принимает два аргумента классов T и Long, производит с ними некоторое действие и ничего не возвращает;

ObjIntConsumer<T> - операция, которая принимает два аргумента классов T и Integer, производит с ними некоторое действие и ничего не возвращает.

Что такое StringJoiner?

Класс StringJoiner используется, чтобы создать последовательность строк, разделенных разделителем с возможностью присоединить к полученной строке префикс и суффикс:

```
StringJoiner joiner = new StringJoiner(".", "prefix-", "-suffix");
for (String s : "Hello the brave world".split(" ")) {
  joiner.add(s);
}
System.out.println(joiner); //prefix-Hello.the.brave.world-suffix
```

Что такое default методы интрефейса?

Java 8 позволяет добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово default:

```
interface Example {
  int process(int a);
  default void show() {
```

```
    System.out.println("default show()");
  }
}
```

Если класс реализует интерфейс, он может, но не обязан, реализовать методы по-умолчанию, уже реализованные в интерфейсе. Класс наследует реализацию по умолчанию.

Если некий класс реализует несколько интерфейсов, которые имеют одинаковый метод по умолчанию, то класс должен реализовать метод с совпадающей сигнатурой самостоятельно. Ситуация аналогична, если один интерфейс имеет метод по умолчанию, а в другом этот же метод является абстрактным - никакой реализации по умолчанию классом не наследуется.

Метод по умолчанию не может переопределить метод класса java.lang.Object.

Помогают реализовывать интерфейсы без страха нарушить работу других классов.

Позволяют избежать создания служебных классов, так как все необходимые методы могут быть представлены в самих интерфейсах.

Дают свободу классам выбрать метод, который нужно переопределить.

Одной из основных причин внедрения методов по умолчанию является возможность коллекций в Java 8 использовать лямбда-выражения.

Как вызывать default метод интерфейса в реализующем этот интерфейс классе?

Используя ключевое слово super вместе с именем интерфейса:

```
interface Paper {
  default void show() {
    System.out.println("default show()");
  }
}

class Licence implements Paper {
  public void show() {
    Paper.super.show();
  }
}
```

Что такое static метод интерфейса?

Статические методы интерфейса похожи на методы по умолчанию, за исключением того, что для них отсутствует возможность переопределения в классах, реализующих интерфейс.

Статические методы в интерфейсе являются частью интерфейса без возможности использовать их для объектов класса реализации;

Методы класса java.lang.Object нельзя переопределить как статические;

Статические методы в интерфейсе используются для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.

Как вызывать static метод интерфейса?

Используя имя интерфейса:

```
interface Paper {
  static void show() {
    System.out.println("static show()");
```

```
  }
}

class Licence {
  public void showPaper() {
    Paper.show();
  }
}
```

Что такое Optional?

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

```
Optional<String> optional = Optional.of("hello");

optional.isPresent(); // true
optional.ifPresent(s -> System.out.println(s.length())); // 5
optional.get(); // "hello"
optional.orElse("ops..."); // "hello"
```

Что такое Stream?

Интерфейс java.util.Stream представляет собой последовательность элементов, над которой можно производить различные операции.

Операции над стримами бывают или промежуточными (intermediate) или конечными (terminal). Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же стрим. Таким образом вы можете строить цепочки из несколько операций над одним и тем же стримом.

У стрима может быть сколько угодно вызовов промежуточных операций и последним вызов конечной операции. При этом все промежуточные операции выполняются лениво и пока не будет вызвана конечная операция никаких действий на самом деле не происходит (похоже на создание объекта Thread или Runnable, без вызова start()).

Стримы создаются на основе источников каких-либо, например классов из java.util.Collection.

Ассоциативные массивы (maps), например, HashMap, не поддерживаются.

Операции над стримами могут выполняться как последовательно, так и параллельно.

Потоки не могут быть использованы повторно. Как только была вызвана какая-нибудь конечная операция, поток закрывается.

Кроме универсальных объектных существуют особые виды стримов для работы с примитивными типами данных int, long и double: IntStream, LongStream и DoubleStream. Эти примитивные стримы работают так же, как и обычные объектные, но со следующими отличиями:

- используют специализированные лямбда-выражения, например, IntFunction или IntPredicate вместо Function и Predicate;
- поддерживают дополнительные конечные операции sum(), average(), mapToObj().

Какие существуют способы создания стрима?

Из коллекции:

```
Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();
```

Из набора значений:

```
Stream<String> fromValues = Stream.of("x", "y", "z");
```

Из массива:

```
Stream<String> fromArray = Arrays.stream(new String[]{"x", "y", "z"});
```

Из файла (каждая строка в файле будет отдельным элементом в стриме):

```
Stream<String> fromFile = Files.lines(Paths.get("input.txt"));
```

Из строки:

```
IntStream fromString = "0123456789".chars();
```

С помощью Stream.builder():

```
Stream<String> fromBuilder = Stream.builder().add("z").add("y").add("z").build();
```

С помощью Stream.iterate() (бесконечный):

```
Stream<Integer> fromIterate = Stream.iterate(1, n -> n + 1);
```

С помощью Stream.generate() (бесконечный):

```
Stream<String> fromGenerate = Stream.generate(() -> "0");
```

В чем разница между Collection и Stream?

Коллекции позволяют работать с элементами по-отдельности, тогда как стримы так делать не позволяют, но вместо этого предоставляют возможность выполнять функции над данными как над одним целым.

Также стоит отметить важность самой концепции сущностей: Collection - это прежде всего воплощение Структуры Данных. Например, Set не просто хранит в себе элементы, он реализует идею множества с уникальными элементами, тогда как Stream, это прежде всего абстракция необходимая для реализации конвеера вычислений, собственно поэтому, результатом работы конвеера являются те или иные Структуры Данных или же результаты проверок/поиска и т.п.

Для чего нужен метод collect() в стримах?

Метод collect() является конечной операцией, которая используется для представление результата в виде коллекции или какой-либо другой структуры данных.

collect() принимает на вход Collector<Тип_источника, Тип_аккумулятора, Тип_результата>, который содержит четыре этапа: supplier - инициализация аккумулятора, accumulator - обработка каждого элемента, combiner - соединение двух аккумуляторов при параллельном выполнении, [finisher]

- необязательный метод последней обработки аккумулятора. В Java 8 в классе Collectors реализовано несколько распространённых коллекторов:

toList(), toCollection(), toSet() - представляют стрим в виде списка, коллекции или множества;
toConcurrentMap(), toMap() - позволяют преобразовать стрим в Map;
averagingInt(), averagingDouble(), averagingLong() - возвращают среднее значение;
summingInt(), summingDouble(), summingLong() - возвращает сумму;
summarizingInt(), summarizingDouble(), summarizingLong() - возвращают SummaryStatistics с разными агрегатными значениями;
partitioningBy() - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>;
groupingBy() - разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>;
mapping() - дополнительные преобразования значений для сложных Collector-ов.

Так же существует возможность создания собственного коллектора через Collector.of():

```
Collector<String, List<String>, List<String>> toList = Collector.of(
  ArrayList::new,
  List::add,
  (l1, l2) -> { l1.addAll(l2); return l1; }
);
```

Для чего в стримах применяются методы forEach() и forEachOrdered()?

forEach() применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется;
forEachOrdered() применяет функцию к каждому объекту стрима с сохранением порядка элементов.

Для чего в стримах предназначены методы map() и mapToInt(), mapToDouble(), mapToLong()?

Метод map() является промежуточной операцией, которая заданным образом преобразует каждый элемент стрима.

mapToInt(), mapToDouble(), mapToLong() - аналоги map(), возвращающие соответствующий числовой стрим (то есть стрим из числовых примитивов):

```
Stream
  .of("12", "22", "4", "444", "123")
  .mapToInt(Integer::parseInt)
  .toArray(); //[12, 22, 4, 444, 123]
```

Какова цель метода filter() в стримах?

Метод filter() является промежуточной операцией принимающей предикат, который фильтрует все элементы, возвращая только те, что соответствуют условию.

Для чего в стримах предназначен метод limit()?

Метод limit() является промежуточной операцией, которая позволяет ограничить выборку определенным количеством первых элементов.

Для чего в стримах предназначен метод sorted()?

Метод sorted() является промежуточной операцией, которая позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator.

Порядок элементов в исходной коллекции остается нетронутым - sorted() всего лишь создает его отсортированное представление.

Для чего в стримах предназначены методы flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong()?

Метод flatMap() похож на map, но может создавать из одного элемента несколько. Таким образом, каждый объект будет преобразован в ноль, один или несколько других объектов, поддерживаемых потоком. Наиболее очевидный способ применения этой операции — преобразование элементов контейнера при помощи функций, которые возвращают контейнеры.

```
Stream
  .of("H e l l o", "w o r l d !")
  .flatMap((p) -> Arrays.stream(p.split(" ")))
  .toArray(String[]::new);//["H", "e", "l", "l", "o", "w", "o", "r", "l", "d", "!"]
```

flatMapToInt(), flatMapToDouble(), flatMapToLong() - это аналоги flatMap(), возвращающие соответствующий числовой стрим.

Расскажите о параллельной обработке в Java 8.

Стримы могут быть последовательными и параллельными. Операции над последовательными стримами выполняются в одном потоке процессора, над параллельными — используя несколько потоков процессора. Параллельные стримы используют общий ForkJoinPool доступный через статический ForkJoinPool.commonPool() метод. При этом, если окружение не является многоядерным, то поток будет выполняться как последовательный. Фактически применение параллельных стримов сводится к тому, что данные в стримах будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются конечные операции.

Для создания параллельного потока из коллекции можно также использовать метод parallelStream() интерфейса Collection.

Чтобы сделать обычный последовательный стрим параллельным, надо вызвать у объекта Stream метод parallel(). Метод isParallel() позволяет узнать является ли стрим параллельным.

С помощью, методов parallel() и sequential() можно определять какие операции могут быть параллельными, а какие только последовательными. Так же из любого последовательного стрима можно сделать параллельный и наоборот:

```
collection
.stream()
.peek(...) // операция последовательна
.parallel()
.map(...) // операция может выполняться параллельно,
.sequential()
.reduce(...) // операция снова последовательна
```

Как правило, элементы передаются в стрим в том же порядке, в котором они определены в источнике данных. При работе с параллельными стримами система сохраняет порядок следования элементов. Исключение составляет метод forEach(), который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод forEachOrdered().

Критерии, которые могут повлиять на производительность в параллельных стримах:

Размер данных - чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.

Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.

Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из ArrayList легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных. А вот коллекция типа LinkedList - не лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.

Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов.

Крайне не рекомендуется использовать параллельные стримы для скольких-нибудь долгих операций (например, сетевых соединений), так как все параллельные стримы работают 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 и тем самым увеличить производительность, использовав промежуточную операцию unordered():

```
collection.parallelStream()
  .sorted()
  .unordered()
  .collect(Collectors.toList());
```

Какие конечные методы работы со стримами вы знаете?

findFirst() возвращает первый элемент;
findAny() возвращает любой подходящий элемент;
collect() представление результатов в виде коллекций и других структур данных;
count() возвращает количество элементов;
anyMatch() возвращает true, если условие выполняется хотя бы для одного элемента;
noneMatch() возвращает true, если условие не выполняется ни для одного элемента;
allMatch() возвращает true, если условие выполняется для всех элементов;
min() возвращает минимальный элемент, используя в качестве условия Comparator;
max() возвращает максимальный элемент, используя в качестве условия Comparator;
forEach() применяет функцию к каждому объекту (порядок при параллельном выполнении не гарантируется);
forEachOrdered() применяет функцию к каждому объекту с сохранением порядка элементов;
toArray() возвращает массив значений;
reduce()позволяет выполнять агрегатные функции и возвращать один результат.

Для числовых стримов дополнительно доступны:

sum() возвращает сумму всех чисел;
average() возвращает среднее арифметическое всех чисел.

Какие промежуточные методы работы со стримами вы знаете?

filter() отфильтровывает записи, возвращая только записи, соответствующие условию;
skip() позволяет пропустить определённое количество элементов в начале;
distinct() возвращает стрим без дубликатов (для метода equals());
map() преобразует каждый элемент;
peek() возвращает тот же стрим, применяя к каждому элементу функцию;
limit() позволяет ограничить выборку определенным количеством первых элементов;
sorted() позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator;
mapToInt(), mapToDouble(), mapToLong() - аналоги map() возвращающие стрим числовых примитивов;
flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong() - похожи на map(), но могут создавать из одного элемента несколько.

Для числовых стримов дополнительно доступен метод mapToObj(), который преобразует числовой стрим обратно в объектный.

Как вывести на экран 10 случайных чисел, используя forEach()?

```
(new Random())
  .ints()
  .limit(10)
  .forEach(System.out::println);
```

Как можно вывести на экран уникальные квадраты чисел используя метод map()?

```
Stream
  .of(1, 2, 3, 2, 1)
  .map(s -> s * s)
  .distinct()
  .forEach(System.out::println);
```

Как вывести на экран количество пустых строк с помощью метода filter()?

```
System.out.println(
  Stream
    .of("Hello", "", ", ", "world", "!")
    .filter(String::isEmpty)
    .count());
```

Как вывести на экран 10 случайных чисел в порядке возрастания?

```
(new Random())
  .ints()
  .limit(10)
  .sorted()
  .forEach(System.out::println);
```

Как найти максимальное число в наборе?

```
Stream
   .of(5, 3, 4, 55, 2)
   .mapToInt(a -> a)
   .max()
   .getAsInt(); //55
```

Как найти минимальное число в наборе?

```
Stream
   .of(5, 3, 4, 55, 2)
   .mapToInt(a -> a)
   .min()
   .getAsInt(); //2
```

Как получить сумму всех чисел в наборе?

```
Stream
   .of(5, 3, 4, 55, 2)
   .mapToInt()
   .sum(); //69
```

Как получить среднее значение всех чисел?

```
Stream
   .of(5, 3, 4, 55, 2)
   .mapToInt(a -> a)
   .average()
   .getAsDouble(); //13.8
```

Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8?

putIfAbsent() добавляет пару «ключ-значение», только если ключ отсутствовал:

```
map.putIfAbsent("a", "Aa");
```

forEach() принимает функцию, которая производит операцию над каждым элементом:

```
map.forEach((k, v) -> System.out.println(v));
```

compute() создаёт или обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

```
map.compute("a", (k, v) -> String.valueOf(k).concat(v)); //["a", "aAa"]
```

computeIfPresent() если ключ существует, обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

```
map.computeIfPresent("a", (k, v) -> k.concat(v));
```

computeIfAbsent() если ключ отсутствует, создаёт его со значением, которое вычисляется (возможно использовать ключ):

```
map.computeIfAbsent("a", k -> "A".concat(k)); //["a","Aa"]
```

getOrDefault() в случае отсутствия ключа, возвращает переданное значение по-умолчанию:

```
map.getOrDefault("a", "not found");
```

merge() принимает ключ, значение и функцию, которая объединяет передаваемое и текущее значения. Если под заданным ключем значение отсутствует, то записывает туда передаваемое значение.

```
map.merge("a", "z", (value, newValue) -> value.concat(newValue)); //["a","Aaz"]
```

Что такое LocalDateTime?

LocalDateTime объединяет вместе LocaleDate и LocalTime, содержит дату и время в календарной системе ISO-8601 без привязки к часовому поясу. Время хранится с точностью до наносекунды. Содержит множество удобных методов, таких как plusMinutes, plusHours, isAfter, toSecondOfDay и т.д.

Что такое ZonedDateTime?

java.time.ZonedDateTime — аналог java.util.Calendar, класс с самым полным объемом информации о временном контексте в календарной системе ISO-8601. Включает временную зону, поэтому все операции с временными сдвигами этот класс проводит с её учётом.

Как получить текущую дату с использованием Date Time API из Java 8?

```
LocalDate.now();
```

Как добавить 1 неделю, 1 месяц, 1 год, 10 лет к текущей дате с использованием Date Time API?

```
LocalDate.now().plusWeeks(1);
LocalDate.now().plusMonths(1);
LocalDate.now().plusYears(1);
LocalDate.now().plus(1, ChronoUnit.DECADES);
```

Как получить следующий вторник используя Date Time API?

```
LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
```

Как получить вторую субботу текущего месяца используя Date Time API?

```
LocalDate
  .of(LocalDate.now().getYear(), LocalDate.now().getMonth(), 1)
  .with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY))
  .with(TemporalAdjusters.next(DayOfWeek.SATURDAY));
```

Как получить текущее время с точностью до миллисекунд используя Date Time API?

```
new Date().toInstant();
```

Как получить текущее время по местному времени с точностью до миллисекунд используя Date Time API?

```
LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault());
```

Как определить повторяемую аннотацию?

Чтобы определить повторяемую аннотацию, необходимо создать аннотацию-контейнер для списка повторяемых аннотаций и обозначить повторяемую мета-аннотацией @Repeatable:

```
@interface Schedulers
{
   Scheduler[] value();
}

@Repeatable(Schedulers.class)
@interface Scheduler
{
   String birthday() default "Jan 8 1935";
}
```

Что такое Nashorn?

Nashorn - это движок JavaScript, разрабатываемый на Java компанией Oracle. Призван дать возможность встраивать код JavaScript в приложения Java. В сравнении с Rhino, который поддерживается Mozilla Foundation, Nashorn обеспечивает от 2 до 10 раз более высокую производительность, так как он компилирует код и передает байт-код виртуальной машине Java непосредственно в памяти. Nashorn умеет компилировать код JavaScript и генерировать классы Java, которые загружаются специальным загрузчиком. Так же возможен вызов кода Java прямо из JavaScript.

Что такое jjs?

jjs это утилита командной строки, которая позволяет исполнять программы на языке JavaScript прямо в консоли.

Какой класс появился в Java 8 для кодирования/декодирования данных?

Base64 - потокобезопасный класс, который реализует кодировщик и декодировщик данных, используя схему кодирования base64 согласно RFC 4648 и RFC 2045.

Base64 содержит 6 основных методов:

getEncoder()/getDecoder() - возвращает кодировщик/декодировщик base64, соответствующий стандарту RFC 4648; getUrlEncoder()/getUrlDecoder() - возвращает URL-safe кодировщик/декодировщик base64, соответствующий стандарту RFC 4648; getMimeEncoder()/getMimeDecoder() - возвращает MIME кодировщик/декодировщик, соответствующий стандарту RFC 2045.

Как создать Base64 кодировщик и декодировщик?

```
// Encode
String b64 = Base64.getEncoder().encodeToString("input".getBytes("utf-8")); //aW5wdXQ==
// Decode
new String(Base64.getDecoder().decode("aW5wdXQ=="), "utf-8"); //input
```

Потоки ввода/вывода в JAVA

В чём 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 кэшируются. 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. Тоже 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 особенности NIO вы знаете?

Каналы и селекторы: NIO поддерживает различные типы каналов. Канал является абстракцией объектов более низкого уровня файловой системы (например, отображенные в памяти файлы и блокировки файлов), что позволяет передавать данные с более высокой скоростью. Каналы не блокируются и поэтому Java предоставляет еще такие инструменты, как селектор, который позволяет выбрать готовый канал для передачи данных, и сокет, который является инструментом для блокировки.

Буферы: имеет буферизация для всех классов-обёрток примитивов (кроме Boolean). Появился абстрактный класс Buffer, который предоставляет такие операции, как clear, flip, mark и т.д. Его подклассы предоставляют методы для получения и установки данных.

Кодировки: появились кодеры и декодеры для отображения байт и символов Unicode. к оглавлению

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 приёмниками 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.

Какие существуют виды потоков ввода/вывода? 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 компрессированных 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;
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 представляет механизм «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 откры-

тых 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;
PrintStream - выходной поток, включающий методы print() и println();
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?

Прежде всего, 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)

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 свои 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 определённое 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.).

Так же 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.

Какие классы поддерживают чтение и запись потоков в компрессированном формате?

DeflaterOutputStream - компрессия данных в формате deflate.
Deflater - компрессия данных в формат ZLIB
ZipOutputStream - потомок DeflaterOutputStream для компрессии данных в формат 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.

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 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 имён и различных атрибутов;
- свободны от некоторых ограничений, присущих жёстким ссылкам (последние действуют только в пределах одной файловой системы (одного раздела) и не могут ссылаться на каталоги).

Сериализация

Что такое «сериализация»?

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.

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 повлечёт 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.

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.

Как создать собственный протокол сериализации?

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.

Когда мы не объявляем serialVersionUID в нашем 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 мы 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.

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»);
  }
}
```

Так же 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.

Многопоточность

Расскажите о модели памяти Java?

Модель памяти Java (Java Memory Model, JMM) описывает поведение потоков в среде исполнения Java. Это часть семантики языка Java, набор правил, описывающий выполнение многопоточных программ и правил, по которым потоки могут взаимодействовать друг с другом посредством основной памяти.

Формально модель памяти определяет набор действий межпоточного взаимодействия (эти действия включают в себя, в частности, чтение и запись переменной, захват и освобождений монитора, чтение и запись volatile переменной, запуск нового потока), а также модель памяти определяет отношение между этими действиями -happens-before - абстракции 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.

Существует несколько основных правил для отношения happens-before:

В рамках одного потока любая операция happens-before любой операцией следующей за ней в исходном коде;
Освобождение монитора (unlock) happens-before захват того же монитора (lock);
Выход из synchronized блока/метода happens-before вход в synchronized блок/метод на том же мониторе;
Запись volatile поля happens-before чтение того же самого volatile поля;
Завершение метода run() экземпляра класса Thread happens-before выход из метода join() или возвращение false методом isAlive() экземпляром того же потока;
Вызов метода start() экземпляра класса Thread happens-before начало метода run() экземпляра того же потока;
Завершение конструктора happens-before начало метода finalize() этого класса;
Вызов метода interrupt() на потоке happens-before обнаружению потоком факта, что данный метод был вызван либо путем выбрасывания исключения InterruptedException, либо с помощью методов isInterrupted() или interrupted().
Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.
Освобождение/захват монитора и запись/чтение в volatile переменную связаны отношением happens-before, только если операции проводятся над одним и тем же экземпляром объекта.
В отношении happens-before участвуют только два потока, о поведении остальных потоков ничего сказать нельзя, пока в каждом из них не наступит отношение happens-before с другим потоком.

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

Видимость (visibility). Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, 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 главной 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. Надо отметить, что это описание не требование к реализации, а всего лишь модель, которая объясняет поведение программы,

так, в качестве локальной памяти не обязательно выступает кэш память, это могут быть регистры процессора или потоки могут вообще не иметь локальной памяти.

При входе в synchronized метод или блок поток обновляет содержимое локальной памяти, а при выходе из synchronized метода или блока поток записывает изменения, сделанные в локальной памяти, в главную. Такое поведение synchronized методов и блоков следует из правил для отношения «происходит раньше»: так как все операции с памятью происходят раньше освобождения монитора и освобождение монитора происходит раньше захвата монитора, то все операции с памятью, которые были сделаны потоком до выхода из synchronized блока должны быть видны любому потоку, который входит в synchronized блок для того же самого монитора. Очень важно, что это правило работает только в том случае, если потоки синхронизируются, используя один и тот же монитор!

Что касается volatile переменных, то запись таких переменных производится в основную память, минуя локальную. и чтение volatile переменной производится также из основной памяти, то есть значение переменной не может сохраняться в регистрах или локальной памяти потока и операция чтения этой переменной гарантированно вернёт последнее записанное в неё значение.

Также модель памяти определяет дополнительную семантику ключевого слова final, имеющую отношение к видимости: после того как объект был корректно создан, любой поток может видеть значения его final полей без дополнительной синхронизации. «Корректно создан» означает, что ссылка на создающийся объект не должна использоваться до тех пор, пока не завершился конструктор объекта. Наличие такой семантики для ключевого слова final позволяет создание неизменяемых (immutable) объектов, содержащих только final поля, такие объекты могут свободно передаваться между потоками без обеспечения синхронизации при передаче.

Есть одна проблема, связанная с final полями: реализация разрешает менять значения таких полей после создания объекта (это может быть сделано, например, с использованием механизма reflection). Если значение final поля—константа, чьё значение известно на момент компиляции, изменения такого поля могут не иметь эффекта, так-как обращения к этой переменной могли быть заменены компилятором на константу. Также спецификация разрешает другие оптимизации, связанные с final полями, например, операции чтения final переменной могут быть переупорядочены с операциями, которые потенциально могут изменить такую переменную. Так что рекомендуется изменять final поля объекта только внутри конструктора, в противном случае поведение не специфицировано.

Reordering (переупорядочивание). Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Вернее, с точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде. Тот же эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, 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 кэши синхронизируются с основной памятью. Еще одна причина reordering, может заключаться в том, что процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее.

Вопрос reordering также регулируется набором правил для отношения «происходит раньше» и у этих правил есть следствие, касающееся порядка операций, используемое на практике: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не-volatile переменных. Это следствие делает возможным использование volatile переменной как флага, сигнализирующем об окончании какого-либо действия. В остальном правила, касающиеся порядка выполнения операций, гарантируют упорядоченность операций для конкретного набора случаев (таких как, например, захват и освобождение монитора), во всех остальных случаях оставляя компилятору и процессору полную свободу для оптимизаций.

Что такое «потокобезопасность»?

Потокобезопасность – свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как предполагается. Например потокобезопасный счётчик не пропустит ни один счёт, даже если один и тот же экземпляр этого счётчика будет использоваться несколькими потоками.

В чём разница между «конкуренцией» и «параллелизмом»?

Конкуренция — это способ одновременного решения множества задач.

Признаки:

Наличие нескольких потоков управления (например Thread в Java, корутина в Kotlin), если поток управления один, то конкурентного выполнения быть не может

Недетерминированный результат выполнения. Результат зависит от случайных событий, реализации и того как была проведена синхронизация. Даже если каждый поток полностью детерминированный, итоговый результат будет недетерминированным

Параллелизм — это способ выполнения разных частей одной задачи.

Признаки:

Необязательно имеет несколько потоков управления

Может приводить к детерминированному результату, так, например, результат умножения каждого элемента массива на число, не изменится, если умножать его по частям параллельно.

Что такое «кооперативная многозадачность»? Какой тип многозадачности использует Java? Чем обусловлен этот выбор?

Кооперативная многозадачность - это способ деления процессорного времени между потоками, при котором каждый поток обязан отдавать управление следующему добровольно.

Преимущества такого подхода - простота реализации, меньшие накладные расходы на переключение контекста.

Недостатки - если один поток завис или ведет себя некорректно, то зависает целиком вся система и другие потоки никогда не получат управление.

Java использует вытесняющую многозадачность, при которой решение о переключении между потоками процесса принимает операционная система.

В отличие от кооперативной многозадачности управление операционной системе передаётся вне зависимости от состояния работающих приложений, благодаря чему, отдельные зависшие потоки процесса, как правило, не «подвешивают» всю систему целиком. За счёт регулярного переключения между задачами также улучшается отзывчивость приложения и повышается оперативность освобождения ресурсов, которые больше не используются.

В реализации вытесняющая многозадачность отличается от кооперативной, в частности, тем, что требует обработки системного прерывания от аппаратного таймера.

Что такое ordering, as-if-serial semantics, sequential consistency, visibility, atomicity, happens-before, mutual exclusion, safe publication?

ordering механизм, который определяет, когда один поток может увидеть out-of-order (неверный) порядок исполнения инструкций другого потока. CPU для для повышения производительности может переупорядочивать процессорные инструкции и выполнять их в произвольном порядке до тех пор пока для потока внутри не будет видно никаких отличий. Гарантия предоставляемая этим механизмом называется as-if-serial semantics.

sequential consistency - то же что и as-if-serial semantics, гарантия того, что в рамках одного потока побочные эффекты от всех операций будут такие, как будто все операции выполняются последовательно.

visibility определяет, когда действия в одном потоке становятся видны из другого потока.

happens-before - логическое ограничение на порядок выполнения инструкций программы. Если указывается, что запись в переменную и последующее ее чтение связаны через эту зависимость, то как бы при выполнении не переупорядочивались инструкции, в момент чтения все связанные с процессом записи результаты уже зафиксированы и видны.

atomicity — атомарность операций. Атомарная операция выглядит единой и неделимой командой процессора, которая может быть или уже выполненной или ещё невыполненной.

mutual exclusion (взаимоисключающая блокировка, семафор с одним состоянием) - механизм, гарантирующий потоку исключительный доступ к ресурсу. Используется для предотвращения одновременного доступа к общему ресурсу. В каждый момент времени таким ресурсом может владеть только один поток. Простейший пример: synchronized(obj) { … }.

safe publication? - показ объектов другим потокам из текущего, не нарушая ограничений visibility. Способы такой публикации в Java:

static{} инициализатор;
volatile переменные;
atomic переменные;
сохранение в разделяемой переменной, корректно защищенной с использованием synchronized(), синхронизаторов или других конструкций, создающих read/write memory barrier;
final переменные в разделяемом объекте, который был корректно проинициализирован.

Чем отличается процесс 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 же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Поток(thread) — определенный 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 процессе выполнения работы выгоднее создавать дополнительные потоки и избегать создания новых процессов.

Что такое «зелёные потоки» и есть ли они в Java?

Зелёные (легковесные) потоки(green threads) - потоки эмулируемые виртуальной машиной или средой исполнения. Создание зелёного потока не подразумевает под собой создание реального потока ОС.

Виртуальная машина Java берёт на себя заботу о переключении между разными green threads, а сама машина работает как один поток ОС. Это даёт несколько преимуществ. Потоки ОС относительно дороги в большинстве POSIX-систем. Кроме того, переключение между native threads гораздо медленнее, чем между green threads.

Это всё означает, что в некоторых ситуациях green threads гораздо выгоднее, чем native threads. Система может поддерживать гораздо большее количество green threads, чем потоков ОС. Например, гораздо практичнее запускать новый green thread для нового HTTP-соединения к веб-серверу, вместо создания нового native thread.

Однако есть и недостатки. Самый большой заключается в том, что вы не можете исполнять два потока одновременно. Поскольку существует только один native thread, только он и вызывается планировщиком ОС. Даже если у вас несколько процессоров и несколько green threads, только один процессор может вызывать green thread. И всё потому, что с точки зрения планировщика заданий ОС всё это выглядит одним потоком.

Начиная с версии 1.2 Java поддерживает native threads, и с тех пор они используются по умолчанию.

Каким образом можно создать поток?

Создать потомка класса Thread и переопределить его метод run();
Создать объект класса Thread, передав ему в конструкторе экземпляр класса, реализующего интерфейс Runnable. Эти интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().
Вызвать метод submit() у экземпляра класса реализующего интерфейс ExecutorService, передав ему в качестве параметра экземпляр класса реализующего интерфейс Runnable или Callable (содержит метод call(), в котором описывается логика выполнения).

Чем различаются Thread и 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 логически отделить логику выполнения задачи от непосредственного управления потоком.

В чём заключается разница между методами start() и run()?

Несмотря на то, что start() вызывает метод run() внутри себя, это не то же самое, что просто вызов run(). Если run() вызывается как обычный метод, то он вызывается в том же потоке и никакой новый поток не запускается, как это происходит, в случае, когда вы вызываете метод start().

Как принудительно запустить поток?

Никак. В Java не существует абсолютно никакого способа принудительного запуска потока. Это контролируется JVM и Java не предоставляет никакго API для управления этим процессом.

Что такое «монитор» в Java?

Монитор, мьютекс (mutex) – это средство обеспечения контроля за доступом к ресурсу. У монитора может быть максимум один владелец в каждый текущий момент времени. Следовательно, если кто-то использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой, желающий использовать тот же ресурс, должен подождать освобождения монитора, захватить его и только потом начать использовать ресурс.

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

В Java у каждого экземпляра объекта есть монитор, который контролируется непосредственно виртуальной машиной. Используется он так: любой нестатический synchronized-метод при своем вызове прежде всего пытается захватить монитор того объекта, у которого он вызван (на который он может сослаться как на this). Если это удалось – метод исполняется. Если нет – поток останавливается и ждет, пока монитор будет отпущен.

Дайте определение понятию «синхронизация».

Синхронизация это процесс, который позволяет выполнять потоки параллельно.

В Java все объекты имеют одну блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта. Если поток получил блокировку, ни один другой поток не может войти в синхронизированный код, пока блокировка не будет снята. Когда поток, владеющий блокировкой, выходит из синхронизированного кода, блокировка снимается. Теперь другой поток может получить блокировку объекта и выполнить синхронизированный код. Если поток пытается получить блокировку объекта, когда другой поток владеет блокировкой, поток переходит в состояние Блокировки до тех пор, пока блокировка не снимется.

Какие существуют способы синхронизации в Java?

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() (опять же, 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!

Системная синхронизация с использованием join(). Метод join(), вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связаный с этим

экземпляром, закончит работу.

Использование классов из пакета java.util.concurrent, который предоставляет набор классов для организации межпоточного взаимодействия. Примеры таких классов - Lock, Semaphore и пр.. Концепция данного подхода заключается в использовании атомарных операций и переменных.

В каких состояниях может находиться поток?

Потоки могут находиться в одном из следующих состояний:

Новый (New). После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым.

Работоспособный (Runnable). Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован. Когда поток находится в этом состоянии, он считается живым.

Работающий (Running). Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его как работающий в данный момент.

Живой, но не работоспособный (Alive, but not runnable). Поток может быть живым, но не работоспособным по нескольким причинам:

Ожидание (Waiting). Поток переходит в состояние Ожидания, вызывая метод wait(). Вызов notify() или notifyAll() может перевести поток из состояния Ожидания в состояние Работоспособный.

Сон (Sleeping). Метод sleep() переводит поток в состояние Сна на заданный промежуток времени в миллисекундах.

Блокировка (Blocked). Поток может перейти в это состояние, в ожидании ресурса, такого как ввод/вывод или из-за блокировки другого объекта. В этом случае поток переходит в состояние Работоспособный, когда ресурс становится доступен.

Мёртвый (Dead). Поток считается мёртвым, когда его метод run() полностью выполнен. Мёртвый поток не может перейти ни в какое другое состояние, даже если для него вызван метод start().

Можно ли создавать новые экземпляры класса, пока выполняется static synchronized метод?

Да, можно создавать новые экземпляры класса, так как статические поля не принадлежат к экземплярам класса.

Зачем может быть нужен private мьютекс?

Объект для синхронизации делается private, чтобы сторонний код не мог на него синхронизироваться и случайно получить взаимную блокировку.

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

Эти методы пределены у класса Object и предназначены для взаимодействия потоков между собой при межпоточной синхронизации.

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 Ожидания (Waiting). Метод notify() подаёт 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.

Поток может вызвать методы wait() или notify() для определённого объекта, только если он в данный момент имеет блокировку на этот объект. wait(), notify() и notifyAll() должны вызываться только из синхронизированного кода.

В чем разница между notify() и notifyAll()?

Дело в том, что «висеть» на методе wait() одного монитора могут сразу несколько потоков. При вызове notify() только один из них выходит из wait() и пытается захватить монитор, а затем продолжает работу со следующего после wait() оператора. Какой из них выйдет - заранее неизвестно. А при вызове notifyAll(), все висящие на wait() потоки выходят из wait(), и все они пытаются захватить монитор. Понятно, что в любой момент времени монитор может быть захвачен только одним потоком, а остальные ждут своей очереди. Порядок очереди определяется планировщиком потоков Java.

Почему методы wait() и notify() вызываются только в синхронизированном блоке?

Монитор надо захватывать в явном виде (через synchronized-блок), потому что методы wait() и notify() не синхронизированы.

Чем отличается работа метода wait() с параметром и без параметра?

wait()

без параметров освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()/notifyAll(),
с параметрами заставит поток ожидать заданное количество времени или вызова notify()/notifyAll().

Чем отличаются методы Thread.sleep() и Thread.yield()?

Метод yield() служит причиной того, что поток переходит из состояния работающий (running) в состояние работоспособный (runnable), давая возможность другим потокам активизироваться. Но следующий выбранный для запуска поток может и не быть другим.

Метод sleep() вызывает засыпание текущего потока на заданное время, состояние изменяется с работающий (running) на ожидающий (waiting).

Как работает метод Thread.join()?

Когда поток вызывает join() для другого потока, текущий работающий поток будет ждать, пока другой поток, к которому он присоединяется, не будет завершён:

void join()
void join(long millis)
void join(long millis, int nanos)

Что такое deadlock?

Взаимная блокировка (deadlock) - явление при котором все потоки находятся в режиме ожидания. Происходит, когда достигаются состояния:

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

Простейший способ избежать взаимной блокировки – не допускать цикличного ожидания. Этого можно достичь, получая мониторы разделяемых ресурсов в определённом 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.

Реальный пример livelock, – когда два 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.

Как проверить, удерживает ли поток монитор определённого ресурса?

Метод Thread.holdsLock(lock) возвращает true, когда текущий поток удерживает монитор у определённого объекта.

На каком объекте происходит синхронизация при вызове static synchronized метода?

У синхронизированного статического метода нет доступа к this, но есть доступ к объекту класса Class, он присутствует в единственном экземпляре и именно он выступает в качестве монитора для синхронизации статических методов. Таким образом, следующая конструкция:

```
public class SomeClass {

  public static synchronized void someMethod() {
    //code
  }
}
```

эквивалентна такой:

```
public class SomeClass {
```

```
  public static void someMethod(){
    synchronized(SomeClass.class){
      //code
    }
  }
}
```

Для чего используется ключевое слово volatile, synchronized, transient, native?

volatile - этот модификатор вынуждает потоки отключить оптимизацию доступа и использовать единственный экземпляр переменной. Если переменная примитивного типа – этого будет достаточно для обеспечения потокобезопасности. Если же переменная является ссылкой на объект – синхронизировано будет исключительно значение этой ссылки. Все же данные, содержащиеся в объекте, синхронизированы не будут!

synchronized - это зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода.

Ключевые слова transient и native к многопоточности никакого отношения не имеют, первое используется для указания полей класса, которые не нужно сериализовать, а второе - сигнализирует о том, что метод реализован в платформо-зависимом коде.

В чём различия между volatile и Atomic переменными?

volatile принуждает использовать единственный экземпляр переменной, но 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.

В чём заключаются различия между java.util.concurrent.Atomic*.compareAndSwap() и java.util.concurrent.Atomic*.weakCompareAndSwap().

weakCompareAndSwap() не создает memory barrier и не дает гарантии happens-before;
weakCompareAndSwap() сильно зависит от кэша/CPU, и может возвращать false без видимых причин;
weakCompareAndSwap(), более легкая, но поддерживаемая далеко не всеми архитектурами и не всегда эффективная операция.

Что значит «приоритет потока»?

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда 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_PRlORITY = 5.

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

Что такое «потоки-демоны»?

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. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода main(), не обращая внимания на то, что поток-демон еще работает.

Можно ли сделать основной поток программы демоном?

Нет. Потоки-демоны позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них.

Что значит «усыпить» поток?

Это значит приостановить его на определенный промежуток времени, вызвав в ходе его выполнения статический метод Thread.sleep() передав в качестве параметра необходимое количество времени в миллисекундах. До истечения этого времени поток может быть выведен из состояния ожидания вызовом interrupt() с выбрасыванием InterruptedException.

Чем отличаются два интерфейса Runnable и Callable?

Интерфейс Runnable появиля 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.call() возвращает объект Future, который может содержать результат вычислений;

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 Java приложении. 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 ещё не завершено. Объекты FutureTask могут быть использованы для обёртки объектов Callable и Runnable. Так как FutureTask реализует Runnable, его можно передать в Executor на выполнение.

В чем заключаются различия между CyclicBarrier и CountDownLatch?

CountDownLatch (замок с обратным отсчетом) предоставляет возможность любому количеству

потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. В конструктор CountDownLatch(int count) обязательно передается количество операций, которое должно быть выполнено, чтобы замок «отпустил» заблокированные потоки.

Примером CountDownLatch из жизни может служить сбор экскурсионной группы: пока не наберется определенное количество человек, экскурсия не начнется.

CyclicBarrier реализует шаблон синхронизации «Барьер». Циклический барьер является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется. Как только все потоки прибыли, выполняется опционное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются». В конструкторы барьера CyclicBarrier(int parties) и CyclicBarrier(int parties, Runnable barrierAction) обязательно передается количество сторон, которые должны «встретиться», и, опционально, действие, которое должно произойти, когда стороны встретились, но перед тем когда они будут «отпущены».

CyclicBarrier является альтернативой метода join(), который «собирает» потоки только после того, как они выполнились.

CyclicBarrier похож на CountDownLatch, но главное различие между ними в том, что использовать «замок» можно лишь единожды - после того, как его счётчик достигнет нуля, а «барьер» можно использовать неоднократно, даже после того, как он «сломается».

Что такое race condition?

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

Существует ли способ решения проблемы race condition?

Распространённые способы решения:

Использование локальной копии — копирование разделяемой переменной в локальную переменную потока. Этот способ работает только тогда, когда переменная одна и копирование производится атомарно (за одну машинную команду), использование volatile.

Синхронизация - операции над разделяемым ресурсом происходят в синхронизированном блоке (при использовании ключевого слова synchronized).

Комбинирование методов - вышеперечисленные способы можно комбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимает ограничение на атомарность, с другой — позволяет избавиться от слишком больших синхронизированных блоков.

Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок — правильное проектирование многозадачной системы.

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 сле-

дующих версиях JDK все они были помечены как deprecated из-за потенциальных угроз взаимной блокировки).

Для корректной остановки потока можно использовать метод класса Thread - interrupt(). Этот метод выставляет некоторый внутренний флаг-статус прерывания. В дальнейшем состояние этого флага можно проверить с помощью метода isInterrupted() или Thread.interrupted() (для текущего потока). Метод interrupt() также 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 – долговременная блокировка крайне маловероятна и тогда можно просто дождаться выхода из метода read(). Если же чтение каким-то образом связано с сетью – стоит использовать неблокирующий ввод-вывод из Java NIO.

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 определённых методов извне. Методика действия при этом остаётся прежней – проверять установку флагов и принимать решения при их изменении. Недостатки такого подхода. Во-первых, потоки в состоянии ожидания таким способом не «оживить». Во-вторых, 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 недетерминированном 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 переведёт 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?

Что происходит, когда в потоке выбрасывается исключение?

Если исключение не поймано – поток «умирает» (переходит в состяние мёртв (dead)).
Если установлен обработчик непойманных исключений, то он возьмёт управление на себя.

Thread.UncaughtExceptionHandler – интерфейс, определённый как вложенный интерфейс для других обработчиков, вызываемых, когда поток внезапно останавливается из-за непойманного исключения. В случае, если поток собирается остановиться из-за непойманного исключения, JVM проверяет его на наличие UncaughtExceptionHandler, используя Thread.getUncaughtExceptionHandler(), и если такой обработчик найдет, то вызовет у него метод uncaughtException(), передав этот поток и исключение в виде аргументов.

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.

Что такое «пул потоков»?

Создание потока является затратной по времени и ресурсам операцией. Количество потоков, которое может быть запущено в рамках одного процесса также ограниченно. Чтобы избежать этих проблем и в целом управлять множеством потоков более эффективно в Java был реализован механизм пула потоков (thread pool), который создаётся во время запуска приложения и в дальнейшем потоки для обработки запросов берутся и переиспользуются уже из него. Таким образом, появляется возможность не терять потоки, сбалансировать приложение по количеству потоков и частоте их создания.

Начиная с Java 1.5 Java API предоставляет фреймворк Executor, который позволяет создавать различные типы пула потоков:

Executor - упрощенный интерфейс пула, содержит один метод для передачи задачи на выполнение;
ExecutorService - расширенный интерфейс пула, с возможностью завершения всех потоков;
AbstractExecutorService - базовый класс пула, реализующий интерфейс ExecutorService;
Executors - фабрика объектов связанных с пулом потоков, в том числе позволяет создать основные типы пулов;
ThreadPoolExecutor - пул потоков с гибкой настройкой, может служить базовым классом для нестандартных пулов;
ForkJoinPool - пул для выполнения задач типа ForkJoinTask;
... и другие.

Методы Executors для создания пулов:

newCachedThreadPool() - если есть свободный поток, то задача выполняется в нем, иначе добавляется новый поток в пул. Потоки не используемые больше минуты завершаются и удалются и кэша. Размер пула неограничен. Предназначен для выполнения множество небольших асинхронных задач;
newCachedThreadPool(ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;
newFixedThreadPool(int nThreads) - создает пул на указанное число потоков. Если новые задачи добавлены, когда все потоки активны, то они будут сохранены в очереди для выполнения позже.

Если один из потоков завершился из-за ошибки, на его место будет запущен другой поток. Потоки живут до тех пор, пока пул не будет закрыт явно методом shutdown().

newFixedThreadPool(int nThreads, ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newSingleThreadScheduledExecutor() - однопотоковый пул с возможностью выполнять задачу через указанное время или выполнять периодически. Если поток был завершен из-за каких-либо ошибок, то для выполнения следующей задачи будет создан новый поток.

newSingleThreadScheduledExecutor(ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newScheduledThreadPool(int corePoolSize) - пул для выполнения задач через указанное время или переодически;

newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

unconfigurableExecutorService(ExecutorService executor) - обертка на пул, запрещающая изменять его конфигурацию;

Какого размера должен быть пул потоков?

Настраивая размер пула потоков, важно избежать двух ошибок: слишком мало потоков (очередь на выполнение будет расти, потребляя много памяти) или слишком много потоков (замедление работы всей систему из-за частых переключений контекста).

Оптимальный размер пула потоков зависит от количества доступных процессоров и природы задач в рабочей очереди. На N-процессорной системе для рабочей очереди, которая будет выполнять исключительно задачи с ограничением по скорости вычислений, можно достигнуть максимального использования CPU с пулом потоков, в котором содержится N или N+1 поток. Для задач, которые могут ждать осуществления I/O (ввода - вывода) - например, задачи, считывающей HTTP-запрос из сокета – может понадобиться увеличение размера пула свыше количества доступных процессоров, потому, что не все потоки будут работать все время. Используя профилирование, можно оценить отношение времени ожидания (WT) ко времени обработки (ST) для типичного запроса. Если назвать это соотношение WT/ST, то для N-процессорной системе понадобится примерно N*(1 + WT/ST) потоков для полной загруженности процессоров.

Использование процессора – не единственный фактор, важный при настройке размера пула потоков. По мере возрастания пула потоков, можно столкнуться с ограничениями планировщика, доступной памяти, или других системных ресурсов, таких, как количество сокетов, дескрипторы открытого файла, или каналы связи базы данных.

Что будет, если очередь пула потоков уже заполнена, но подаётся новая задача?

Если очередь пула потоков заполнилась, то поданная задача будет «отклонена». Например - метод submit() у ThreadPoolExecutor выкидывает RejectedExecutionException, после которого вызывается RejectedExecutionHandler.

В чём заключается различие между методами submit() и execute() у пула потоков?

Оба метода являются способами подачи задачи в пул потоков, но между ними есть небольшая разница.

execute(Runnable command) определён в интерфейсе Executor и выполняет поданную задачу и ничего не возвращает.

submit() – перегруженный метод, определённый в интерфейсе ExecutorService. Способен принимать задачи типов Runnable и Callable и возвращать объект Future, который можно использовать для контроля и управления процессом выполнения, получения его результата.

В чем заключаются различия между стеком (stack) и кучей (heap) с точки зрения многопоточности?

Стек – участок памяти, тесно связанный с потоками. У каждого потока есть свой стек, которые хранит локальные переменные, параметры методов и стек вызовов. Переменная, хранящаяся в стеке одного потока, не видна для другого.

Куча – общий участок памяти, который делится между всеми потоками. Объекты, неважно локальные или любого другого уровня, создаются в куче. Для улучшения производительности, поток обычно кэширует значения из кучи в свой стек, в этом случае для того, чтобы указать потоку, что переменную следует читать из кучи используется ключевое слово volatile.

Как поделиться данными между двумя потоками?

Данными между потоками возможно делиться, используя общий объект или параллельные структуры данных, например BlockingQueue.

Какой параметр запуска JVM используется для контроля размера стека потока?

-Xss

Как получить дамп потока?

Среды исполнения Java на основе HotSpot генерируют только дамп в формате HPROF. В распоряжении разработчика имеется несколько интерактивных методов генерации дампов и один метод генерации дампов на основе событий.

Интерактивные методы:

Использование Ctrl+Break: если для исполняющегося приложения установлена опция командной строки -XX:+HeapDumpOnCtrlBreak, то дамп формата HPROF генерируется вместе с дампом потока при наступлении события Ctrl+Break или SIGQUIT (обычно генерируется с помощью kill -3), которое инициируется посредством консоли. Эта опция может быть недоступна в некоторых версиях. В этом случае можно попытаться использовать следующую опцию: -Xrunhprof:format=b,file=heapdump.hprof

Использование инструмента jmap: утилита jmap, поставляемая в составе каталога /bin/ комплекта JDK, позволяет запрашивать дамп HPROF из исполняющегося процесса.

Использование операционной системы: Для создания файла ядра можно воспользоваться неразрушающей командой gcore или разрушающими командами kill -6 или kill -11. Затем извлечь дамп кучи из файла ядра с помощью утилиты jmap.

Использование инструмента JConsole. Операция dumpHeap предоставляется в JConsole как MBean-компонент HotSpotDiagnostic. Эта операция запрашивает генерацию дампа в формате HPROF.

Метод на основе событий:

Событие OutOfMemoryError: Если для исполняющегося приложения установлена опция командной строки -XX:+HeapDumpOnOutOfMemoryError, то при возникновении ошибки

OutOfMemoryError генерируется дамп формата HPROF. Это идеальный метод для «production» систем, поскольку он практически обязателен для диагностирования проблем памяти и не сопровождается постоянными накладными расходами с точки зрения производительности. В старых выпусках сред исполнения Java на базе HotSpot для этого события не устанавливается предельное количество дампов кучи в пересчете на одну JVM; в более новых выпусках допускается не более одного дампа кучи для этого события на каждый запуск JVM.

Что такое ThreadLocal-переменная?

ThreadLocal - класс, позволяющий имея одну переменную, иметь различное её значение для каждого из потоков.

У каждого потока - т.е. экземпляра класса Thread - есть ассоциированная с ним таблица ThreadLocal-переменных. Ключами таблицы являются ссылки на объекты класса ThreadLocal, а значениями - ссылки на объекты, «захваченные» ThreadLocal-переменными, т.е. ThreadLocal-переменные отличаются от обычных переменных тем, что у каждого потока свой собственный, индивидуально инициализируемый экземпляр переменной. Доступ к значению можно получить через методы get() или set().

Например, если мы объявим ThreadLocal-переменную: ThreadLocal<Object> locals = new ThreadLocal<Object>();. А затем, в потоке, сделаем locals.set(myObject), то ключом таблицы будет ссылка на объект locals, а значением - ссылка на объект myObject. При этом для другого потока существует возможность «положить» внутрь locals другое значение.

Следует обратить внимание, что ThreadLocal изолирует именно ссылки на объекты, а не сами объекты. Если изолированные внутри потоков ссылки ведут на один и тот же объект, то возможны коллизии.

Так же важно отметить, что т.к. ThreadLocal-переменные изолированы в потоках, то инициализация такой переменной должна происходить в том же потоке, в котором она будет использоваться. Ошибкой является инициализация такой переменной (вызов метода set()) в главном потоке приложения, потому как в данном случае значение, переданное в методе set(), будет «захвачено» для главного потока, и при вызове метода get() в целевом потоке будет возвращен null.

Назовите различия между synchronized и ReentrantLock?

В Java 5 появился интерфейс Lock предоставляющий возможности более эффективного и тонкого контроля блокировки ресурсов. ReentrantLock – распространённая реализация Lock, которая предоставляет Lock с таким же базовым поведением и семантикой, как у synchronized, но расширенными возможностями, такими как опрос о блокировании (lock polling), ожидание блокирования заданной длительности и прерываемое ожидание блокировки. Кроме того, он предлагает гораздо более высокую эффективность функционирования в условиях жесткой состязательности.

Что понимается под блокировкой с повторным входом (reentrant)? Просто то, что есть подсчет сбора данных, связанный с блокировкой, и если поток, который удерживает блокировку, снова ее получает, данные отражают увеличение, и тогда для реального разблокирования нужно два раза снять блокировку. Это аналогично семантике synchronized; если поток входит в синхронный блок, защищенный монитором, который уже принадлежит потоку, потоку будет разрешено дальнейшее функционирование, и блокировка не будет снята, когда поток выйдет из второго (или последующего) блока synchronized, она будет снята только когда он выйдет из первого блока synchronized, в который он вошел под защитой монитора.

Lock lock = new ReentrantLock();

```
lock.lock();
try {
 // update object state
}
finally {
 lock.unlock();
}
```

Реализация ReentrantLock гораздо более масштабируема в условиях состязательности, чем реализация synchronized. Это значит, что когда много потоков соперничают за право получения блокировки, общая пропускная способность обычно лучше у ReentrantLock, чем у synchronized. JVM требуется меньше времени на установление очередности потоков и больше времени на непосредственно выполнение.

У ReentrantLock (как и у других реализаций Lock) блокировка должна обязательно сниматься в finally блоке (иначе, если бы защищенный код выбросил исключение, блокировка не была бы снята). Используя синхронизацию, JVM гарантирует, что блокировка автоматически снимаются.

Резюмируя можно сказать, что когда состязания за блокировку нет либо оно очень мало, то synchronized возможно будет быстрее. Если присутствует заметное состязание за доступ к ресурсу, то скорее всего ReentrantLock даст некое преимущество.

Что такое ReadWriteLock?

ReadWriteLock – это интерфейс расширяющий базовый интерфейс Lock. Используется для улучшения производительности в многопоточном процессе и оперирует парой связанных блокировок (одна - для операций чтения, другая - для записи). Блокировка чтения может удерживаться одновременно несколькими читающими потоками, до тех пор пока не появится записывающий. Блокировка записи является эксклюзивеной.

Существует реализующий интерфейс ReadWriteLock класс ReentrantReadWriteLock, который поддерживает до 65535 блокировок записи и до стольки же блокировок чтения.

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();
Lock rLock = rwLock.readLock();
Lock wLock = rwLock.writeLock();

wLock.lock();
try {
   // exclusive write
} finally {
   wLock.unlock();
}

rLock.lock();
try {
   // shared reading
} finally {
   rLock.unlock();
}
```

Что такое «блокирующий метод»?

Блокирующий метод – метод, который блокируется, до тех пор, пока задание не выполнится, например метод accept() у ServerSocket блокируется в ожидании подключения клиента. Здесь блоки-

рование означает, что контроль не вернётся к вызывающему методу до тех пор, пока не выполнится задание. Так же существуют асинхронные или неблокирующиеся методы, которые могут завершится до выполнения задачи.

Что такое «фреймворк Fork/Join»?

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) идёт процесс «свёртки» - 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 всё меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из 1 элемента. Хотя в некотором смысле Join будет необходим и тут, т.к. всё равно остаётся необходимость дождаться пока не закончится выполнение всех подзадач.

Ещё одно замечательное 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 всё ещё заняты.

Что такое Semaphore?

Semaphore – это новый тип синхронизатора: семафор со счётчиком, реализующий шаблон синхронизации Семафор. Доступ управляется с помощью счётчика: изначальное значение счётчика задаётся в конструкторе при создании синхронизатора, когда поток заходит в заданный блок кода, то значение счётчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счётчика 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 такое double checked locking Singleton?

double checked locking Singleton - это один из способов создания потокобезопасного класса реализующего шаблон Одиночка. Данный метод пытается оптимизировать производительность, блокируясь только случае, когда экземпляр одиночки создаётся впервые.

```
class DoubleCheckedLockingSingleton {
  private static volatile DoubleCheckedLockingSingleton instance;

  static DoubleCheckedLockingSingleton getInstance() {
    DoubleCheckedLockingSingleton current = instance;
    if (current == null) {
      synchronized (DoubleCheckedLockingSingleton.class) {
        current = instance;

        if (current == null) {
```

```
                instance = current = new DoubleCheckedLockingSingleton();
            }
        }
    }
    return current;
  }
}
```

Следует заметить, что требование volatile обязательно. Проблема Double Checked Lock заключается в модели памяти Java, точнее в порядке создания объектов, когда возможна ситуация, при которой другой поток может получить и начать использовать (на основании условия, что указатель не нулевой) не полностью сконструированный объект. Хотя эта проблема была частично решена в JDK 1.5, однако рекомендация использовать voloatile для Double Checked Lock остаётся в силе.

Как создать потокобезопасный Singleton?

Static field

```
public class Singleton {
        public static final Singleton INSTANCE = new Singleton();
}
```

Enum

```
public enum Singleton {
        INSTANCE;
}
```

Synchronized Accessor

```
public class Singleton {
        private static Singleton instance;

        public static synchronized Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }
}
```

Double Checked Locking & volatile

```
public class Singleton {
    private static volatile Singleton instance;

    public static Singleton getInstance() {
                Singleton localInstance = instance;
                if (localInstance == null) {
                        synchronized (Singleton.class) {
                                localInstance = instance;
                                if (localInstance == null) {
                                        instance = localInstance = new Singleton();
                                }
```

```
                }
            }
            return localInstance;
        }
}
```

On Demand Holder Idiom

```
public class Singleton {

    public static class SingletonHolder {
        public static final Singleton HOLDER_INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.HOLDER_INSTANCE;
    }
}
```

Чем полезны неизменяемые объекты?

Неизменяемость (immutability) помогает облегчить написание многопоточного кода. Неизменяемый объект может быть использован без какой-либо синхронизации. К сожалению в Java нет аннотации @Immutable, которая делает объект неизменяемым, для этого разработчикам нужно самим создавать класс с необходимыми характеристиками. Для этого необходимо следовать некоторым общим принципам: инициализация всех полей только конструкторе, отсутствие методов setX() вносящих изменения в поля класса, отсутствие утечек ссылки, организация отдельного хранилища копий изменяемых объектов и т.д.

Что такое busy spin?

busy spin – это техника, которую программисты используют, чтобы заставить поток ожидать при определённом условии. В отличие от традиционных методов wait(), sleep() или yield(), которые подразумевают уступку процессорного времени, этот метод вместо уступки выполняет пустой цикл. Это необходимо, для того, чтобы сохранить кэш процессора, т.к. в многоядерных системах, существует вероятность, что приостановленный поток продолжит своё выполнение уже на другом ядре, а это повлечет за собой перестройку состояния процессорного кэша, которая является достаточно затратной процедурой.

Перечислите принципы, которым вы следуете в многопоточном программировании?

При написании многопоточных программ следует придерживаться определённых правил, которые помогают обеспечить достойную производительность приложения в сочетании с удобной отладкой и простотой дальнейшей поддержки кода.

Всегда давайте значимые имена своим потокам. Процесс отладки, нахождения ошибок или отслеживание исключения в многопоточном коде – довольно сложная задача. OrderProcessor, QuoteProcessor или TradeProcessor намного информативнее, чем Thread1, Thread2 и Thread3. Имя должно отражать задачу, выполняемую данным потоком.

Избегайте блокировок или старайтесь уменьшить масштабы синхронизации. Блокировка затратна, а переключение контекста ещё более ресурсоёмко. Пытайтесь избегать синхронизации и блокировки насколько это возможно, и организуйте критическую секцию в минимально необходимом

объёме. Поэтому синхронизированный блок всегда предпочительней синхронизированного метода, дополнительно наделяя возможностью абсолютного контроля над масштабом блокировки.

Обрабатывайте прерывание потока с особой тщательностью. Нет ничего хуже оставшегося заблокированным ресурса или системы в неконстистентном, по причине неподтверждённой транзакции, состоянии.

Помните об обработке исключений. Выброшенные InterruptedException должны быть адекватно обработаны, а не просто подавлены. Так же не стоит пренебрегать Thread.UncaughtExceptionHandler. При использовании пула потоков необходимо помнить, что он зачастую просто «проглатывает» исключения. Так, если вы отправили на выполнение Runnable нужно обязательно поместить код выполнения задачи внутрь блока try-catch. Если в очередь пула помещается Callable, необходимо удостоверится, что результат выполнения всегда изымается помощью блокирующего get(), чтобы в случае возникновения существовала возможнотсь заново выбросить произошедшее исключение.

Между синхронизаторами и wait() и notify() следует выбирать синхронизаторы. Во-первых, синхронизаторы, типа CountDownLatch, Semaphore, CyclicBarrier или Exchanger упрощают написание кода. Очень сложно реализовывать комплексный управляющий поток, используя wait() и notify(). Во-вторых, эти классы написаны и поддерживаются настоящими мастерами своего дела и есть шанс, что в последующих версиях JDK они будут оптимизированы изнутри или заменены более производительной внешней реализацией.

Почти всегда использование Concurrent collection выгоднее использования Synchronized collection, т.к. первые более современны (используют все доступные на момент их написания новшества языка) и масштабируемы, чем их синхронизированые аналоги.

Какое из следующих утверждений о потоках неверно?

Если метод start() вызывается дважды для одного и того же объекта Thread, во время выполнения генерируется исключение.

Порядок, в котором запускались потоки, может не совпадать с порядком их фактического выполнения.

Если метод run() вызывается напрямую для объекта Thread, во время выполнения генерируется исключение.

Если метод sleep() вызывается для потока, во время выполнения синхронизированного кода, блокировка не снимается.

Правильный ответ: 3. Если метод run() вызывается напрямую для объекта Thread, во время выполнения исключение не генерируется. Однако, код, написанный в методе run() будет выполняться текущим, а не новым потоком. Таким образом, правильный способ запустить поток – это вызов метода start(), который приводит к выполнению метода run() новым потоком.

Вызов метода start() дважды для одного и того же объекта Thread приведёт к генерированию исключения IllegalThreadStateException во время выполнения, следовательно, утверждение 1 верно. Утверждение 2 верно, так как порядок, в котором выполняются потоки, определяется Планировщиком потоков, независимо от того, какой поток запущен первым. Утверждение 4 верно, так как поток не освободит блокировки, которые он держит, когда он переходит в состояние Ожидания.

Даны 3 потока T1, T2 и T3? Как реализовать выполнение в последовательности T1, T2, T3?

Такой последовательности выполнения можно достичь многими способами, например просто воспользоваться методом join(), чтобы запустить поток в момент, когда другой уже закончит своё выполнение. Для реализации заданной последовательности, нужно запустить последний поток первым, и затем вызывать метод join() в обратном порядке, то есть T3 вызывает T2.join, а T2 вызывает T1.join, таким образом T1 закончит выполнение первым, а T3 последним.

Напишите минимальный неблокирующий стек (всего два метода — push() и pop()).

```
class NonBlockingStack<T> {
  private final AtomicReference<Element> head = new AtomicReference<>(null);

  Stack<T> push(final T value) {
    final Element current = new Element();
    current.value = value;
    Element recent;
    do {
      recent = head.get();
      current.previous = recent;
    } while (!head.compareAndSet(recent, current));
    return this;
  }

  T pop() {
    Element result;
    Element previous;
    do {
      result = head.get();
      if (result == null) {
        return null;
      }
      previous = result.previous;
    } while (!head.compareAndSet(result, previous));
    return result.value;
  }

  private class Element {
    private T value;
    private Element previous;
  }
}
```

Напишите минимальный неблокирующий стек (всего два метода — push() и pop()) с использованием Semaphore.

```
class SemaphoreStack<T> {
  private final Semaphore semaphore = new Semaphore(1);
  private Node<T> head = null;

  SemaphoreStack<T> push(T value) {
    semaphore.acquireUninterruptibly();
    try {
      head = new Node<>(value, head);
    } finally {
      semaphore.release();
    }

    return this;
  }
```

```
  T pop() {
    semaphore.acquireUninterruptibly();
    try {
      Node<T> current = head;
      if (current != null) {
        head = head.next;
        return current.value;
      }
      return null;
    } finally {
      semaphore.release();
    }
  }

  private static class Node<E> {
    private final E value;
    private final Node<E> next;

    private Node(E value, Node<E> next) {
      this.value = value;
      this.next = next;
    }
  }
}
```

Напишите минимальный неблокирующий ArrayList (всего четыре метода — add(), get(), remove(), size()).

```
class NonBlockingArrayList<T> {
  private volatile Object[] content = new Object[0];

  NonBlockingArrayList<T> add(T item) {
    return add(content.length, item);
  }

  NonBlockingArrayList<T> add(int index, T item) {
    if (index < 0) {
      throw new IllegalArgumentException();
    }
    boolean needsModification = index > content.length - 1;
    if (!needsModification) {
      if (item == null) {
        needsModification = content[index] != null;
      } else {
        needsModification = item.equals(content[index]);
      }
    }
    if (needsModification) {
      final Object[] renewed = Arrays.copyOf(content, Math.max(content.length, index + 1));
      renewed[index] = item;
      content = renewed;
    }
    return this;
```

```
  }

  NonBlockingArrayList<T> remove(int index) {
    if (index < 0 || index >= content.length) {
      throw new IllegalArgumentException();
    }
    int size = content.length - 1;
    final Object[] renewed = new Object[size];
    System.arraycopy(content, 0, renewed, 0, index);
    if (index + 1 < size) {
      System.arraycopy(content, index + 1, renewed, index, size - index);
    }
    content = renewed;
    return this;
  }

  T get(int index) {
    return (T) content[index];
  }

  int size() {
    return content.length;
  }
}
```

Напишите потокобезопасную реализацию класса с неблокирующим методом BigInteger next(), который возвращает элементы последовательности: [1, 2, 4, 8, 16, ...].

```
class PowerOfTwo {
  private AtomicReference<BigInteger> current = new AtomicReference<>(null);

  BigInteger next() {
    BigInteger recent, next;
    do {
      recent = current.get();
      next = (recent == null) ? BigInteger.valueOf(1) : recent.shiftLeft(1);
    } while (!current.compareAndSet(recent, next));
    return next;
  }
}
```

Напишите простейший многопоточный ограниченный буфер с использованием synchronized.

```
class QueueSynchronized<T> {
  private volatile int size = 0;
  private final Object[] content;
  private final int capacity;

  private int out;
  private int in;
```

```
private final Object isEmpty = new Object();
private final Object isFull = new Object();

QueueSynchronized(final int capacity) {
   this.capacity = capacity;
   content = new Object[this.capacity];
   out = 0;
   in = 0;
   size = 0;
}

private int cycleInc(int index) {
   return (++index == capacity)
         ? 0
         : index;
}

@SuppressWarnings(«unchecked»)
T get() throws InterruptedException {
   if (size == 0) {
      synchronized (isEmpty) {
         while (size < 1) {
            isEmpty.wait();
         }
      }
   }
   try {
      synchronized (this) {
         final Object value = content[out];
         content[out] = null;
         if (size > 1) {
            out = cycleInc(out);
         }
         size--;
         return (T) value;
      }
   } finally {
      synchronized (isFull) {
         isFull.notify();
      }
   }
}

QueueSynchronized<T> put(T value) throws InterruptedException {
   if (size == capacity) {
      synchronized (isFull) {
         while (size == capacity) {
            isFull.wait();
         }
      }
   }
   synchronized (this) {
      if (size == 0) {
```

```
            content[in] = value;
        } else {
            in = cycleInc(in);
            content[in] = value;
        }
        size++;
    }
    synchronized (isEmpty) {
        isEmpty.notify();
    }
    return this;
  }
}
```

Напишите простейший многопоточный ограниченный буфер с использованием ReentrantLock.

```
class QueueReentrantLock<T> {

  private volatile int size = 0;
  private final Object[] content;
  private final int capacity;

  private int out;
  private int in;

  private final ReentrantLock lock = new ReentrantLock();
  private final Condition isEmpty = lock.newCondition();
  private final Condition isFull = lock.newCondition();

  QueueReentrantLock(int capacity) {
    try {
      lock.lock();
      this.capacity = capacity;
      content = new Object[capacity];
      out = 0;
      in = 0;
    } finally {
      lock.unlock();
    }
  }

  private int cycleInc(int index) {
    return (++index == capacity)
        ? 0
        : index;
  }

  @SuppressWarnings(«unchecked»)
  T get() throws InterruptedException {
    try {
      lock.lockInterruptibly();
      if (size == 0) {
        while (size < 1) {
```

```
                isEmpty.await();
            }
        }
        final Object value = content[out];
        content[out] = null;
        if (size > 1) {
            out = cycleInc(out);
        }
        size--;
        isFull.signal();
        return (T) value;
    } finally {
        lock.unlock();
    }
}

QueueReentrantLock<T> put(T value) throws InterruptedException {
    try {
        lock.lockInterruptibly();
        if (size == capacity) {
            while (size == capacity) {
                isFull.await();
            }
        }
        if (size == 0) {
            content[in] = value;
        } else {
            in = cycleInc(in);
            content[in] = value;
        }
        size++;
        isEmpty.signal();
    } finally {
        lock.unlock();
    }
    return this;
}
}
```

Servlets, JSP, JSTL

Что такое «сервлет»?

Сервлет является интерфейсом, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ. Хотя сервлеты могут обслуживать любые запросы, они обычно используются для расширения веб-серверов.

Большинство необходимых для создания сервлетов классов и интерфейсов содержатся в пакетах javax.servlet и javax.servlet.http.

Основные методы сервлета:

public void init(ServletConfig config) throws ServletException запускается сразу после загрузки сервлета в память;
public ServletConfig getServletConfig() возвращает ссылку на объект, который предоставляет доступ к информации о конфигурации сервлета;
public String getServletInfo() возвращает строку, содержащую информацию о сервлете, например: автор и версия сервлета;
public void service(ServletRequest request, ServletResponse response) throws ServletException, java.io.IOException вызывается для обработки каждого запроса;
public void destroy() выполняется перед выгрузкой сервлета из памяти.

Текущая спецификация - Servlet 3.1 описана в JSR-340 и принята в 2013 году.

В чем заключаются преимущества технологии сервлетов над CGI (Common Gateway Interface)?

Сервлеты предоставляют лучшую производительность обработки запросов и более эффективное использование памяти за счет использования преимущество многопоточности (на каждый запрос создается новая нить, что быстрее выделения памяти под новый объект для каждого запроса, как это происходит в CGI).

Сервлеты, как платформа и система являются независимыми. Таким образом веб-приложение написанное с использованием сервлетов может быть запущена в любом контейнере сервлетов, реализующим этот стандарт и в любой операционной системе.

Использование сервлетов повышает надежность программы, т.к. контейнер сервлетов самостоятельно заботится о жизненном цикле сервлетов (а значит и за утечками памяти), безопасности и сборщике мусора.

Сервлеты относительно легки в изучении и поддержке, таким образом разработчику необходимо заботиться только о бизнес-логике приложения, а не внутренней реализации веб-технологий.

Какова структура веб-проекта?

src/main/java Исходники приложения/библиотеки

src/main/resources Ресурсные файлы приложения/библиотеки

src/main/filters Файлы сервлетных фильтров

src/main/webapp Исходники веб-приложения

src/test/java Исходники тестов

src/test/resources Ресурсные файлы тестов

src/test/filters Тесты сервлетных фильтров

src/it Интеграционные тесты

src/assembly Описание сборки

src/site Сайт

LICENSE.txt Лицензия проекта

NOTICE.txt Замечания и определения библиотек зависимостей.

README.txt Описание проекта

Что такое «контейнер сервлетов»?

Контейнер сервлетов — программа, представляющая собой сервер, который занимается системной поддержкой сервлетов и обеспечивает их жизненный цикл в соответствии с правилами, определёнными в спецификациях. Может работать как полноценный самостоятельный веб-сервер, быть поставщиком страниц для другого веб-сервера, или интегрироваться в Java EE сервер приложений.

Контейнер сервлетов обеспечивает обмен данными между сервлетом и клиентами, берёт на себя выполнение таких функций, как создание программной среды для функционирующего сервлета, идентификацию и авторизацию клиентов, организацию сессии для каждого из них.

Наиболее известные реализации контейнеров сервлетов:

Apache Tomcat
Jetty
JBoss
WildFly
GlassFish
IBM WebSphere
Oracle Weblogic

Зачем нужны сервера приложений, если есть контейнеры сервлетов?

- Пулы соединений с БД

Возможность периодического тестирования доступности СУБД и обновления соединения в случае восстановления после сбоев

Замена прав доступа при подключении

Балансировка нагрузки между несколькими СУБД, определение доступность или недоступность того или иного узла

Защита пула соединений от некорректного кода в приложении, которое по недосмотру не возвращает соединения, просто отбирая его после какого-то таймаута.

JMS

Доступность сервера очередей сообщений «из-коробки».

Возможность кластеризации очередей, т.е. доступность построения распределенных очередей, расположенных сразу на нескольких серверах, что существенно увеличивает масштабируемость и доступность приложения

Возможность миграции очередей - в случае падения одного из серверов, его очереди автоматически перемещаются на другой, сохраняя необработанные сообщения.

В некоторых серверах приложений поддерживается Unit-of-Order - гарантированный порядок

обработки сообщений, удовлетворяющих некоторым критериям.

- JTA Встроенная поддержка распределенных транзакций для обеспечения согласованности данных в разные СУБД или очереди.
- Безопасность

Наличие множества провайдеров безопасности и аутентификации:

во встроенном или внешнем LDAP-сервере

в базе данных

в различных Internet-directory (специализированных приложениях для управления правами доступа)

Доступность Single-Sign-On (возможности разделения пользовательской сессии между приложениями) посредством Security Assertion Markup Language (SAML) 1/2 или Simple and Protected Negotiate (SPNEGO) и Kerberos: один из серверов выступает в роли базы для хранения пользователей, все другие сервера при аутентификации пользователя обращаются к этой базе.

Возможность авторизации посредством протокола eXtensible Access Control Markup Language (XACML), позволяющего описывать довольно сложные политики (например, приложение доступно пользователю только в рабочее время).

Кластеризация всего вышеперечисленного

- Масштабируемость и высокая доступность Для контейнера сервлетов обычно так же возможно настроить кластеризацию, но она будет довольно примитивной, так как в случае его использования имееются следующие ограничения:

Сложность передачи пользовательской сессии из одного центра обработки данных (ЦоД) в другой через Интернет

Отсутствие возможности эффективно настроить репликации сессий на большом (состоящем из 40-50 экземпляров серверов) кластере

Невозможность обеспечения миграции экземпляров приложения на другой сервер

Недоступность механизмов автоматического мониторинга и реакции на ошибки

- Управляемость

Присутствие единого центра управления, т.н. AdminServer и аналога NodeManager'а, обеспечивающего

Возможность одновременного запуска нескольких экземпляров сервера

Просмотр состояния запущенных экземпляров сервера, обработчиков той или иной очереди, на том или ином сервере, количества соединений с той или иной БД

Административный канал и развертывание в промышленном режиме Некоторые сервера приложений позволяют включить так называемый «административный канал» - отдельный порт, запросы по которому имеют приоритет.

Просмотр состояния (выполняющихся транзакций, потоков, очередей) в случае недоступности («зависания») сервера

Обновление приложений «на-лету», без простоя:

добавление на сервер новой версии приложения в «закрытом» режиме, пока пользователи продолжают работать со предыдущей

тестирование корректности развертывания новой версии

«скрытый» перевод на использование новой версии всех пользователей

Как контейнер сервлетов управляет жизненным циклом сервлета, когда и какие методы вызываются?

Контейнер сервлетов управляет четырьмя фазами жизненного цикла сервлета:

Загрузка класса сервлета — когда контейнер получает запрос для сервлета, то происходит загрузка класса сервлета в память и вызов его конструктора без параметров.

Инициализация класса сервлета — после того как класс загружен контейнер инициализирует объект ServletConfig для этого сервлета и внедряет его через init() метод. Это и есть место где сервлет класс преобразуется из обычного класса в сервлет.

Обработка запросов — после инициализации сервлет готов к обработке запросов. Для каждого

запроса клиента сервлет контейнер порождает новый поток и вызывает метод service() путем передачи ссылки на объекты ответа и запроса.

Удаление - когда контейнер останавливается или останавливается приложение, то контейнер сервлетов уничтожает классы сервлетов путем вызова destroy() метода.

Таким образом, сервлет создаётся при первом обращении к нему и живёт на протяжении всего времени работы приложения (в отличии от объектов классов, которые уничтожаются сборщиком мусора после того как они уже не используются) и весь жизненный цикл сервлета можно описать как последовательность вызова методов:

public void init(ServletConfig config) – используется контейнером для инициализации сервлета. Вызывается один раз за время жизни сервлета.

public void service(ServletRequest request, ServletResponse response) – вызывается для каждого запроса. Метод не может быть вызван раньше выполнения init() метода.

public void destroy() – вызывается для уничтожения сервлета (один раз за время жизни сервлета).

Что такое «дескриптор развертывания»?

Дескриптор развертывания — это конфигурационный файл артефакта, который будет развернут в контейнере сервлетов. В спецификации Java Platform, Enterprise Edition дескриптор развертывания описывает то, как компонент, модуль или приложение (такое, как веб-приложение или приложение предприятия) должно быть развернуто.

Этот конфигурационный файл указывает параметры развертывания для модуля или приложения с определенными настройками, параметры безопасности и описывает конкретные требования к конфигурации. Для синтаксиса файлов дескриптора развертывания используется язык XML.

```
<?xml version=»1.0» encoding=»UTF-8» ?>
<web-app xmlns=»http://java.sun.com/xml/ns/j2ee»
  xmlns:xsi=»http://www.w3.org/2001/XMLSchema-instance»
  xsi:schemaLocation=»http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd»
  version=»2.4»>

  <display-name>Display name.</display-name>
  <description>Description text.</description>

  <servlet>
    <servlet-name>ExampleServlet</servlet-name>
    <servlet-class>xyz.company.ExampleServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
      <param-name>configuration</param-name>
      <param-value>default</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>ExampleServlet</servlet-name>
    <url-pattern>/example</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>ExampleJSP</servlet-name>
```

```
    <jsp-file>/sample/Example.jsp</jsp-file>
  </servlet>

  <context-param>
    <param-name>myParam</param-name>
    <param-value>the value</param-value>
  </context-param>
</web-app>
```

Для веб-приложений дескриптор развертывания должен называться web.xml и находиться в директории WEB-INF, в корне веб-приложения. Этот файл является стандартным дескриптором развертывания, определенным в спецификации. Также есть и другие типы дескрипторов, такие, как файл дескриптора развертывания sun-web.xml, содержащий специфичные для Sun GlassFish Enterprise Server данные для развертывания именно для этого сервера приложений или файл application.xml в директории META-INF для приложений J2EE.

Какие действия необходимо проделать при создании сервлетов?

Чтобы создать сервлет ExampleServlet, необходимо описать его в дескрипторе развёртывания:

```
<servlet-mapping>
  <servlet-name>ExampleServlet</servlet-name>
  <url-pattern>/example</url-pattern>
</servlet-mapping>
<servlet>
  <servlet-name>ExampleServlet</servlet-name>
  <servlet-class>xyz.company.ExampleServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>default</param-value>
  </init-param>
</servlet>
```

Затем создать класс xyz.company.ExampleServlet путём наследования от HttpServlet и реализовать логику его работы в методе service() или методах doGet()/doPost().

В каком случае требуется переопределять метод service()?

Метод service() переопределяется, когда необходимо, чтобы сервлет обрабатывал все запросы (и GET, и POST) в одном методе.

Когда контейнер сервлетов получает запрос клиента, то происходит вызов метода service(), который в зависимости от поступившего запроса вызывает или метод doGet() или метод doPost().

Есть ли смысл определять для сервлета конструктор? Каким образом лучше инициализировать данные?

Большого смысла определять для сервлета конструктор нет, т.к. инициализировать данные лучше не в конструкторе, а переопределив метод init(), в котором имеется возможность доступа к параметрам инициализации сервлета через использование объекта ServletConfig.

Почему необходимо переопределить только init() метод без аргументов?

Метод init() переопределяется, если необходимо инициализировать какие-то данные до того как сервлет начнет обрабатывать запросы.

При переопределении метода init(ServletConfig config), первым должен быть вызван метод super(config), который обеспечит вызов метода init(ServletConfig config) суперкласса. GenericServlet предоставляет другой метод init() без параметров, который будет вызываться в конце метода init(ServletConfig config).

Необходимо использовать переопределенный метод init() без параметров для инициализации данных во избежание каких-либо проблем, например ошибку, когда вызов super() не указан в переопределенном init(ServletConfig config).

Какие наиболее распространенные задачи выполняются в контейнере сервлетов?

Поддержка обмена данными. Контейнер сервлетов предоставляет легкий способ обмена данными между веб клиентом (браузером) и сервлетом. Благодаря контейнеру нет необходимости создавать слушателя сокета на сервере для отслеживания запросов от клиента, а так же разбирать запрос и генерировать ответ. Все эти важные и комплексные задачи решаются с помощью контейнера и разработчик может сосредоточиться на бизнес логике приложения.

Управление жизненным циклом сервлетов и ресурсов. Начиная от загрузки сервлета в память, инициализации, внедрения методов и заканчивая уничтожением сервлета. Контейнер так же предоставляет дополнительные утилиты, например JNDI, для управления пулом ресурсов.

Поддержка многопоточности. Контейнер самостоятельно создает новую нить для каждого запроса и предоставляет ей запрос и ответ для обработки. Таким образом сервлет не инициализируется заново для каждого запроса и тем самым сохраняет память и уменьшает время до обработки запроса.

Поддержка JSP. JSP классы не похожи на стандартные классы джавы, но контейнер сервлетов преобразует каждую JSP в сервлет и далее управляется контейнером как обычным сервлетом.

Различные задачи. Контейнер сервлетов управляет пулом ресурсов, памятью приложения, сборщиком мусора. Предоставляются возможности настройки безопасности и многое другое.

Что вы знаете о сервлетных фильтрах?

Сервлетный фильтр - это Java-код, пригодный для повторного использования и позволяющий преобразовать содержание HTTP-запросов, HTTP-ответов и информацию, содержащуюся в заголовках HTML. Сервлетный фильтр занимается предварительной обработкой запроса, прежде чем тот попадает в сервлет, и/или последующей обработкой ответа, исходящего из сервлета.

Сервлетные фильтры могут:

- перехватывать инициацию сервлета прежде, чем сервлет будет инициирован;
- определить содержание запроса прежде, чем сервлет будет инициирован;
- модифицировать заголовки и данные запроса, в которые упаковывается поступающий запрос;
- модифицировать заголовки и данные ответа, в которые упаковывается получаемый ответ;
- перехватывать инициацию сервлета после обращения к сервлету.

Сервлетный фильтр может быть конфигурирован так, что он будет работать с одним сервлетом или группой сервлетов. Основой для формирования фильтров служит интерфейс javax.servlet.Filter, который реализует три метода:

```
  void init(FilterConfig config) throws ServletException;
  void destroy();
  void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
IOException, ServletException;
```

Метод init() вызывается прежде, чем фильтр начинает работать,и настраивает конфигурационный объект фильтра. Метод doFilter() выполняет непосредственно работу фильтра. Таким образом, сервер вызывает init() один раз, чтобы запустить фильтр в работу, а затем вызывает doFilter() столько раз, сколько запросов будет сделано непосредственно к данному фильтру. После того, как фильтр заканчивает свою работу, вызывается метод destroy().

Интерфейс FilterConfig содержит метод для получения имени фильтра, его параметров инициации и контекста активного в данный момент сервлета. С помощью своего метода doFilter() каждый фильтр получает текущий запрос request и ответ response, а также FilterChain, содержащий список фильтров, предназначенных для обработки. В doFilter() фильтр может делать с запросом и ответом всё, что ему захочется - собирать данные или упаковывать объекты для придания им нового поведения. Затем фильтр вызывает chain.doFilter(), чтобы передать управление следующему фильтру. После возвращения этого вызова фильтр может по окончании работы своего метода doFilter() выполнить дополнительную работу над полученным ответом. К примеру, сохранить регистрационную информацию об этом ответе.

После того, как класс-фильтр откомпилирован, его необходимо установить в контейнер и «приписать» (map) к одному или нескольким сервлетам. Объявление и подключение фильтра отмечается в дескрипторе развёртывания web.xml внутри элементов <filter> и <filter-mapping>. Для подключение фильтра к сервлету необходимо использовать вложенные элементы <filter-name> и <servlet-name>.

Объявление класс-фильтра FilterConnect с именем FilterName:

```
<filter>
    <filter-name>FilterName</filter-name>
    <filter-class>FilterConnect</filter-class>
    <init-param>
        <!--- фильтр имеет параметр инициализации `active`, которому присваивается значение
`true`. -->
        <param-name>active</param-name>
        <param-value>true</param-true>
    </init-param>
</filter>
```

Подключение фильтра FilterName к сервлету ServletName:

```
<filter-mapping>
    <filter-name>FilterName</filter-name>
    <servlet-name>ServletName</servlet-name>
</filter-mapping>
```

Для связи фильтра со страницами HTML или группой сервлетов необходимо использовать тег <url-pattern>:

Подключение фильтра FilterName ко всем вызовам .html страниц

```
<filter-mapping>
    <filter-name>FilterName</filter-name>
```

```
    <url-pattern>*.html</url-pattern>
</filter-mapping>
```

Порядок, в котором контейнер строит цепочку фильтров для запроса определяется следующими правилами:

цепочка, определяемая <url-pattern>, выстраивается в том порядке, в котором встречаются соответствующие описания фильтров в web.xml;
последовательность сервлетов, определенных с помощью <servlet-name>, также выполняется в той последовательности, в какой эти элементы встречаются в дескрипторе развёртывания web.xml.

Зачем в сервлетах используются различные listener?

Listener (слушатель) работает как триггер, выполняя определённые действия при наступлении какого-либо события в жизненном цикле сервлета.

Слушатели, разделённые по области видимости (scope):

Request:
ServletRequestListener используется для того, чтобы поймать момент создания и уничтожения запроса;
ServletRequestAttributeListener используется для прослушивании событий происходящих с атрибутами запроса.
Context:
ServletContextListener позволяет поймать момент, когда контекст инициализируется либо уничтожается;
ServletContextAttributeListener используется для прослушивании событий происходящих с атрибутами в контексте.
Session:
HttpSessionListener позволяет поймать момент создания и уничтожения сессии;
HttpSessionAttributeListener используется при прослушивании событий происходящих с атрибутами в сессии;
HttpSessionActivationListener используется в случае, если происходит миграция сессии между различными JVM в распределённых приложениях;
HttpSessionBindingListener так же используется для прослушивания событий происходящих с атрибутами в сессии. Разница между HttpSessionAttributeListener и HttpSessionBindingListener слушателями: первый декларируется в web.xml; экземпляр класса создается контейнером автоматически в единственном числе и применяется ко всем сессиям; второй: экземпляр класса должен быть создан и закреплён за определённой сессией «вручную», количество экземпляров также регулируется самостоятельно.

Подключение слушателей:

```
<web-app>
  ...
  <listener>
    <listener-class>xyz.company.ExampleListener</listener-class>
  </listener>
  ...
</web-app>
```

HttpSessionBindingListener подключается в качестве атрибута непосредственно в сессию, т.е., чтобы его подключить необходимо:

- создать экземпляр класса реализующего этот интерфейс;
- положить созданный экземпляр в сессию при помощи setAttribute(String, Object).

Когда стоит использовать фильтры сервлетов, а когда слушателей?

Следует использовать фильтры, если необходимо обрабатывать входящие или исходящие данные (например: для аутентификации, преобразования формата, компрессии, шифрования и т.д.), в случае, когда необходимо реагировать на события - лучше применять слушателей.

Как реализовать запуск сервлета одновременно с запуском приложения?

Контейнер сервлетов обычно загружает сервлет по первому запросу клиента.

Если необходимо загрузить сервлет прямо на старте приложения (например если загрузка сервлета происходит длительное время) следует использовать элемент <load-on-startup> в дескрипторе или аннотацию @loadOnStartup в коде сервлета, что будет указывать на необходимость загрузки сервлета при запуске.

Если целочисленное значение этого параметра отрицательно, то сервлет будет загружен при запросе клиента. В противном случае - загрузится на старте приложения, при этом, чем число меньше, тем раньше в очереди на загрузку он окажется.

```
<servlet>
  <servlet-name>ExampleServlet</servlet-name>
  <servlet-class>xyz.company.ExampleServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Как обработать в приложении исключения, выброшенные другим сервлетом?

Когда приложение выбрасывет исключение контейнер сервлетов обрабатывает его и создаёт ответ в формате HTML. Это аналогично тому что происходит при кодах ошибок вроде 404, 403 и т.д.

В дополнении к этому существует возможность написания собственных сервлетов для обработки исключений и ошибок с указанием их в дескрипторе развертывания:

```
<error-page>
  <error-code>404</error-code>
  <location>/AppExceptionHandler</location>
</error-page>

<error-page>
  <exception-type>javax.servlet.ServletException</exception-type>
  <location>/AppExceptionHandler</location>
</error-page>
```

Основная задача таких сервлетов - обработать ошибку/исключение и сформировать понятный ответ пользователю. Например, предоставить ссылку на главную страницу или же описание ошибки.

Что представляет собой ServletConfig?

Интерфейс javax.servlet.ServletConfig используется для передачи сервлету конфигурационной информации. Каждый сервлет имеет свой собственный экземпляр объекта ServletConfig, создаваемый контейнером сервлетов.

Для установки параметров конфигурации используются параметры init-param в web.xml:

```
<servlet>
  <servlet-name>ExampleServlet</servlet-name>
  <servlet-class>xyz.company.ExampleServlet</servlet-class>
  <init-param>
    <param-name>exampleParameter</param-name>
    <param-value>parameterValue</param-value>
  </init-param>
</servlet>
```

или аннотации @WebInitParam:

```
@WebServlet(
  urlPatterns = «/example»,
  initParams = {
    @WebInitParam(name = «exampleParameter», value = «parameterValue»)
  }
)
public class ExampleServlet extends HttpServlet {
  //...
}
```

Для получения ServletConfig сервлета используется метод getServletConfig().

Что представляет собой ServletContext?

Уникальный (в рамках веб-приложения) объект ServletContext реализует интерфейс javax.servlet.ServletContext и предоставляет сервлетам доступ к параметрам этого веб-приложения. Для предоставления доступа используется элемент <context-param> в web.xml:

```
<web-app>
  ...
  <context-param>
    <param-name>exampleParameter</param-name>
    <param-value>parameterValue</param-value>
  </context-param>
  ...
</web-app>
```

Объект ServletContext можно получить с помощью метода getServletContext() у интерфейса ServletConfig. Контейнеры сервлетов так же могут предоставлять контекстные объекты, уникальные для группы сервлетов. Каждая из групп будет связана со своим набором URL-путей хоста. В спецификации Servlet 3 ServletContext был расширен и теперь предоставляет возможности программного добавления слушателей и фильтров в приложение. Так же у этого интерфейса имеется множество полезных методов таких как getServerInfo(), getMimeType(), getResourceAsStream() и т.д.

В чем отличия ServletContext и ServletConfig?

ServletConfig уникален для сервлета, а ServletContext - для приложения;
ServletConfig используется для предоставления параметров инициализации конкретному сервлету, а ServletContext для предоставления параметров инициализации для всех сервлетов приложения;
для ServletConfig возможности модифицировать атрибуты отсутствуют, атрибуты в объекте ServletContext можно изменять.

Для чего нужен интерфейс ServletResponse?

Интерфейс ServletResponse используется для отправки данных клиенту. Все методы данного инструмента служат именно этой цели:

String getCharacterEncoding() - возвращает MIME тип кодировки (к примеру - UTF8), в которой будет выдаваться информация;
void setLocale(Locale locale)/Locale getLocale() - указывают на язык используемый в документе;
ServletOutputStream getOutputStream()/PrintWriter getWriter() - возвращают потоки вывода данных;
void setContentLength(int len) - устанавливает значение поля HTTP заголовка Content-Length;
void setContentType(String type) - устанавливает значение поля HTTP заголовка Content-Type.
void reset() - позволяет сбрасить HTTP заголовок к значениям по-умолчанию, если он ещё не был отправлен
и др.

Для чего нужен интерфейс ServletRequest?

Интерфейс ServletRequest используется для получения параметров соединения, запроса, а также заголовков, входящего потока данных и т.д.

Что такое Request Dispatcher?

Интерфейс RequestDispatcher используется для передачи запроса другому ресурсу, при этом существует возможность добавления данных полученных из этого ресурса к собственному ответу сервлета. Так же этот интерфейс используется для внутренней коммуникации между сервлетами в одном контексте.

В интерфейсе объявлено два метода:

void forward(ServletRequest var1, ServletResponse var2) — передает запрос из сервлета к другому ресурсу (сервлету, JSP или HTML файлу) на сервере.
void include(ServletRequest var1, ServletResponse var2) — включает контент ресурса (сервлет, JSP или HTML страница) в ответ.
Доступ к интерфейсу можно получить с помощью метода интерфейса ServletContext - RequestDispatcher getRequestDispatcher(String path), где путь начинающийся с /, интерпретируется относительно текущего корневого пути контекста.

Как из одного сервлета вызвать другой сервлет?

Для вызова сервлета из того же приложения необходимо использовать механизм внутренней коммуникации сервлетов (inter-servlet communication mechanisms) через вызовы методов RequestDispatcher:

forward() - передаёт выполнение запроса в другой сервлет;
include() - предоставляет возможность включить результат работы другого сервлета в возвращаемый ответ.

Если необходимо вызывать сервлет принадлежащий другому приложению, то использовать RequestDispatcher уже не получится, т.к. он определен только для текущего приложения. Для подобных целей необходимо использовать метод ServletResponse - sendRedirect() которому предоставляется полный URL другого сервлета. Для передачи данных между сервлетами можно использовать cookies.

Чем отличается sendRedirect() от forward()?

forward():

Выполняется на стороне сервера;
Запрос перенаправляется на другой ресурс в пределах того же сервера;
Не зависит от протокола клиентского запроса, так как обеспечивается контейнером сервлетов;
Нельзя применять для внедрения сервлета в другой контекст;
Клиент не знает о фактически обрабатываемом ресурсе и URL в строке остается прежним;
Выполняется быстрее метода sendRedirect();
Определён в интерфейсе RequestDispatcher.

sendRedirect():

Выполняется на стороне клиента;
Клиенту возвращается ответ 302 (redirect) и запрос перенаправляется на другой сервер;
Может использоваться только с клиентами HTTP;
Разрешается применять для внедрения сервлета в другой контекст;
URL адрес изменяется на адрес нового ресурса;
Медленнее forward() т.к. требует создания нового запроса;
Определён в интерфейсе HttpServletResponse.

Для чего используются атрибуты сервлетов и как происходит работа с ними?

Атрибуты сервлетов используются для внутренней коммуникации сервлетов.

В веб-приложении существует возможность работы с атрибутами используя методы setAttribute(), getAttribute(), removeAttribute(), getAttributeNames(), которые предоставлены интерфейсами ServletRequest, HttpSession и ServletContext (для областей видимости request, session, context соответственно).

Каким образом можно допустить в сервлете deadlock?

Можно получить блокировку, например, допустив циклические вызовы метода doPost() в методе doGet() и метода doGet() в методе doPost().

Как получить реальное расположение сервлета на сервере?

Реальный путь к расположению сервлета на сервере можно получить из объекта ServletContext:

getServletContext().getRealPath(request.getServletPath()).

Как получить информацию о сервере из сервлета?

Информацию о сервере можно получить из объекта ServletContext:

getServletContext().getServerInfo().

Как получить IP адрес клиента на сервере?

IP адрес клиента можно получить вызвав request.getRemoteAddr().

Какие классы-обертки для сервлетов вы знаете?

Собственные обработчики ServletRequest и ServletResponse можно реализовать добавив новые или переопределив существующие методы у классов-обёрток ServletRequestWrapper (HttpServletRequestWrapper) и ServletResponseWrapper (HttpServletRequestWrapper).

В чем отличия GenericServlet и HttpServlet?

Абстрактный класс GenericServlet — независимая от используемого протокола реализация интерфейса Servlet, а абстрактный класс HttpServlet в свою очередь расширяет GenericServlet для протокола HTTP..

Почему HttpServlet класс объявлен как абстрактный?

Класс HTTPServlet предоставляет лишь общую реализацию сервлета для HTTP протокола. Реализация ключевых методов doGet() и doPost(), содержащих основную бизнес-логику перекладывается на разработчика и по умолчанию возвращает HTTP 405 Method Not Implemented error.

Какие основные методы присутствуют в классе HttpServlet?

doGet() - для обработки HTTP запросов GET;
doPost() - для обработки HTTP запросов POST;
doPut() - для обработки HTTP запросов PUT;
doDelete() - для обработки HTTP запросов DELETE;
doHead() - для обработки HTTP запросов HEAD;
doOptions() - для обработки HTTP запросов OPTIONS;
doTrace() - для обработки HTTP запросов TRACE.

Стоит ли волноваться о многопоточной безопасности работая с сервлетами?

Методы init() и destroy() вызываются один раз за жизненный цикл сервлета — поэтому по поводу них беспокоиться не стоит.

Методы doGet(), doPost(), service() вызываются на каждый запрос клиента и т.к. сервлеты используют многопоточность, то здесь задумываться о потокобезопасной работе обязательно. При этом правила использования многопоточности остаются теми же: локальные переменные этих методов будут созданы отдельно для каждого потока, а при использовании глобальных разделяемых ресурсов необходимо использовать синхронизацию или другие приёмы многопоточного программирования.

Какой метод HTTP не является неизменяемым?

HTTP метод называется неизменяемым, если он на один и тот же запрос всегда возвращает одинаковый результат. HTTP методы GET, PUT, DELETE, HEAD и OPTIONS являются неизменяемыми, поэтому необходимо реализовывать приложение так, чтобы эти методы возвращали одинаковый результат постоянно. К изменяемым методам относится метод POST, который и используется для реализации чего-либо, что изменяется при каждом запросе.

К примеру, для доступа к статической HTML странице используется метод GET, т.к. он всегда возвращает одинаковый результат. При необходимости сохранять какую-либо информацию, например в базе данных, нужно использовать POST метод.

Какие есть методы отправки данных с клиента на сервер?

GET - используется для запроса содержимого указанного ресурса, изображения или гипертекстового документа. Вместе с запросом могут передаваться дополнительные параметры как часть URI, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер. Этот метод является основным методом взаимодействия браузера клиента и веб-сервера.

POST - используется для передачи пользовательских данных в содержимом HTTP-запроса на сервер. Пользовательские данные упакованы в тело запроса согласно полю заголовка Content-Type и/или включены в URI запроса. При использовании метода POST под URI подразумевается ресурс, который будет обрабатывать запрос.

В чем разница между методами GET и POST?

GET передает данные серверу используя URL, тогда как POST передает данные, используя тело HTTP запроса. Длина URL ограничена 1024 символами, это и будет верхним ограничением для данных, которые можно отослать через GET. POST может отправлять гораздо большие объемы данных. Лимит устанавливается web-server и составляет обычно около 2 Mb.

Передача данных методом POST более безопасна, чем методом GET, так как секретные данные (например пароль) не отображаются напрямую в web-клиенте пользователя, в отличии от URL, который виден почти всегда. Иногда это преимущество превращается в недостаток - вы не сможете послать данные за кого-то другого.

GETметод является неизменяемым, тогда как POST — изменяемый.

В чем разница между PrintWriter и ServletOutputStream?

PrintWriter — класс для работы с символьным потоком, экземпляр которого можно получить через метод ServletResponse getWriter();

ServletOutputStream — класс для работы байтовым потоком. Для получения его экземпляра используется метод ServletResponse getOutputStream().

к оглавлению

Можно ли одновременно использовать в сервлете PrintWriter и ServletOutputStream?

Так сделать не получится, т.к. при попытке одновременного вызова getWriter() и getOutputStream() будет выброшено исключение java.lang.IllegalStateException с сообщением, что уже был вызван другой метод.

Расскажите об интерфейсе SingleThreadModel.

Интерфейс SingleThreadModel является маркерным - в нем не объявлен ни один метод, однако, если сервлет реализует этот интерфейс, то метод service() этого сервлета гарантированно не будет одновременно выполняться в двух потоках. Контейнер сервлетов либо синхронизирует обращения к единственному экземпляру, либо обеспечивает поддержку пула экземпляров и перенаправление запроса свободному сервлету. Другими словами, контейнер гарантирует отсутствие конфликтов при одновременном обращении к переменным или методам экземпляра сервлета. Однако существуют также и другие разделяемые ресурсы, которые даже при использовании этого интерфейса, остаются всё так же доступны обработчикам запросов в других потоках. Т.о. пользы от использования этого интерфейса немного и в спецификации Servlet 2.4 он был объявлен deprecated.

Что означает URL encoding? Как это осуществить в Java?

URL Encoding — процесс преобразования данных в форму CGI (Common Gateway Interface), не содержащую пробелов и нестандартных символов, которые заменяются в процессе кодирования на специальные escape-символы. В Java для кодирования строки используется метод java.net.URLEncoder.encode(String str, String unicode). Обратная операция декодирования возможна через использование метода java.net.URLDecoder.decode(String str, String unicode).

Hello мир! преобразовывается в Hello%20%D0%BC%D0%B8%D1%80!.

Какие различные методы управления сессией в сервлетах вы знаете?

При посещении клиентом Web-ресурса и выполнении вариантов запросов, контекстная информация о клиенте не хранится. В протоколе HTTP нет возможностей для сохранения и изменения информации о предыдущих посещениях клиента. Сеанс (сессия) – соединение между клиентом и сервером, устанавливаемое на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и Web-сервером. Каждый клиент устанавливает с сервером свой собственный сеанс. Сеансы используются для обеспечения хранения данных во время нескольких запросов Web-страницы или на обработку информации, введенной в пользовательскую форму в результате нескольких HTTP-соединений (например, клиент совершает несколько покупок в интернет-магазине; студент отвечает на несколько тестов в системе дистанционного обучения).

Существует несколько способов обеспечения уникального идентификатора сессии:

User Authentication – Предоставление учетных данных самим пользователем в момент аутентификации. Переданная таким образом информация в дальнейшем используется для поддержания сеанса. Это метод не будет работать, если пользователь вошёл в систему одновременно из нескольких мест.

HTML Hidden Field – Присвоение уникального значения скрытому полю HTML страницы, в момент когда пользователь начинает сеанс. Этот метод не может быть использован со ссылками, потому что нуждается в подтверждении формы со скрытым полем каждый раз во время формирования запроса. Кроме того, это не безопасно, т.к. существует возможность простой подмены такого идентификатора.

URL Rewriting – Добавление идентификатора сеанса как параметра URL. Достаточно утомительная операция, потому что требует постоянного отслеживания этого идентификатора при каждом запросе или ответе.

Cookies – Использование небольших фрагментов данных, отправленных web-сервером и хранимых на устройстве пользователя. Данный метод не будет работать, если клиент отключает использование cookies.

Session Management API – Использование специального API для отслеживания сеанса, построенный на основе и на методах описанных выше и который решает частные проблемы перечисленных способов:

Чаще всего недостаточно просто отслеживать сессию, необходимо ещё и сохранять какие-либо дополнительные данные о ней, которые могут потребоваться при обработке последующих запросов. Осуществление такого поведения требует много дополнительных усилий.

Все вышеперечисленные методы не являются универсальными: для каждого из них можно подобрать конкретный сценарий, при котором они не будут работать.

Что такое cookies?

Cookies («куки») — небольшой фрагмент данных, отправленный web-сервером и хранимый на устройстве пользователя. Всякий раз при попытке открыть страницу сайта, web-клиент пересылает соответствующие этому сайту cookies web-серверу в составе HTTP-запроса. Применяется для сохранения данных на стороне пользователя и на практике обычно используется для:

- аутентификации пользователя;
- хранения персональных предпочтений и настроек пользователя;
- отслеживания состояния сеанса доступа пользователя;
- ведения разнообразной статистики.

Какие методы для работы с cookies предусмотрены в сервлетах?

Servlet API предоставляет поддержку cookies через класс javax.servlet.http.Cookie:

Для получения массива cookies из запроса необходимо воспользоваться методом HttpServletRequest.getCookies(). Методов для добавления cookies в HttpServletRequest не предусмотрено.

Для добавления cookie в ответ используется HttpServletResponse.addCookie(Cookie c). Метода получения cookies в HttpServletResponse отсутствует.

Что такое URL Rewriting?

URL Rewriting - специальная перезапись (перекодирование) оригинального URL. Данный механизм может использоваться для управления сессией в сервлетах, когда cookies отключены.

Зачем нужны и чем отличаются методы encodeURL() и encodeRedirectURL()?

HttpServletResponse.encodeURL() предоставляет способ преобразования URL в HTML гиперссылку с преобразованием спецсимволов и пробелов, а так же добавления session id к URL. Такое поведение аналогично java.net.URLEncoder.encode(), но с добавлением дополнительного параметра jsessionid в конец URL.

Метод HttpServletResponse.encodeRedirectURL() преобразует URL для последующего использования в методе sendRedirect().

Таким образом для HTML гиперссылок при URL rewriting необходимо использовать encodeURL(), а для URL при перенаправлении - encodeRedirectUrl().

Что такое «сессия»?

Сессия - это сеанс связи между клиентом и сервером, устанавливаемый на определенное время. Сеанс устанавливается непосредственно между клиентом и веб-сервером в момент получения первого запроса к веб-приложению. Каждый клиент устанавливает с сервером свой собственный сеанс, который сохраняется до окончания работы с приложением.

Как уведомить объект в сессии, что сессия недействительна или закончилась?

Чтобы быть уверенным в том, что объект будет оповещён о прекращении сессии, нужно реализовать интерфейс javax.servlet.http.HttpSessionBindingListener. Два метода этого интерфейса: valueBound() и valueUnbound() используются при добавлении объекта в качестве атрибута к сессии и при уничтожении сессии соответственно.

Какой существует эффективный способ удостоверится, что все сервлеты доступны только для пользователя с верной сессией?

Сервлет фильтры используются для перехвата всех запросов между контейнером сервлетов и сервлетом. Поэтому логично использовать соответствующий фильтр для проверки необходимой информации (например валидности сессии) в запросе.

Как мы можем обеспечить transport layer security для нашего веб приложения?

Для обеспечения transport layer security необходимо настроить поддержку SSL сервлет контейнера. Как это сделать зависит от конкретной реализации сервлет-контейнера.

Как организовать подключение к базе данных, обеспечить журналирование в сервлете?

При работе с большим количеством подключений к базе данных рекомендуется инициализировать их в servlet context listener, а также установить в качестве атрибута контекста для возможности использования другими сервлетами.

Журналирование подключается к сервлету стандартным для логгера способом (например для log4j это может быть property-файл или XML-конфигурация) , а далее эта информация используется при настройке соответствующего context listener.

Какие основные особенности появились в спецификации Servlet 3?

Servlet Annotations. До Servlet 3 вся конфигурация содержалась в web.xml, что приводило к ошибкам и неудобству при работе с большим количестве сервлетов. Примеры аннотаций: @WebServlet, @WebInitParam, @WebFilter, @WebListener.

Web Fragments. Одностраничное веб приложение может содержать множество модулей: все модули прописываются в fragment.xml в папке META-INF\. Это позволяет разделять веб приложение на отдельные модули, собранные как .jar-файлы в отдельной lib\ директории.

Динамическое добавление веб компонентов. Появилась возможность программно добавлять фильтры и слушатели, используя ServletContext объект. Для этого применяются методы addServlet(),

addFilter(), addListener(). Используя это нововведение стало доступным построение динамической системы, в которой необходимый объект будет создан и вызван только по необходимости.

Асинхронное выполнение. Поддержка асинхронной обработки позволяет передать выполнение запроса в другой поток без удержания всего сервера занятым.

Какие способы аутентификации доступны сервлету?

Спецификация сервлетов определяет четыре типа проверки подлинности:

HTTP Basic Authentication - BASIC. При доступе к закрытым ресурсам появится окно, которое попросит ввести данные для аутентификации.

Form Based Login - FORM. Используется собственная html форма:

HTTP Digest Authentication - DIGEST. Цифровая аутентификация с шифрованием.

HTTPS Authentication - CLIENT-CERT. Аутентификация с помощью клиентского сертификата.

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

Что такое Java Server Pages (JSP)?

JSP (JavaServer Pages) — платформонезависимая переносимая и легко расширяемая технология разработки веб-приложений, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница JSP содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP-элементы, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также EL (Expression Language), для внедрения Java-кода в статичное содержимое JSP-страниц.

Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц Jasper, и затем компилируется в байт-код JVM.

JSP-страницы загружаются на сервере и управляются Java EE Web Application. Обычно такие страницы упакованы в файловые архивы .war и .ear.

Зачем нужен JSP?

JSP расширяет технологию сервлетов обеспечивая возможность создания динамических страницы с HTML подобным синтаксисом.

Хотя создание представлений поддерживается и в сервлетах, но большая часть любой веб-страницы является статической, поэтому код сервлета в таком случае получается чересчур перегруженным, замусоренным и поэтому при его написании легко допустить ошибку.

Еще одним преимуществом JSP является горячее развертывание - возможность заменить одну страницу на другую непосредственно в контейнере без необходимости перекомпилировать весь проект или перезапускать сервер.

Однако рекомендуется избегать написания серьёзной бизнес-логики в JSP и использовать страницу только в качестве представления.

Опишите, как обрабатываются JSP страницы, начиная от запроса к серверу, заканчивая ответом пользователю.

Когда пользователь переходит по ссылке на страницу page.jsp, он отправляет http-запрос на сервер GET /page.jsp. Затем, на основе этого запроса и текста самой страницы, сервер генерирует java-класс, компилирует его и выполняет полученный сервлет, формирующий ответ пользователю в виде представления этой страницы, который сервер и перенаправляет обратно пользователю.

Расскажите об этапах (фазах) жизненного цикла JSP.

Если посмотреть код внутри созданной JSP страницы, то он будет выглядеть как HTML и не будет похож на java класс. Конвертацией JSP страниц в HTML код занимается контейнер, который так же создает и сервлет для использования в веб приложении.

Жизненный цикл JSP состоит из нескольких фаз, которыми руководит JSP контейнер:

Translation – проверка и парсинг кода JSP страницы для создания кода сервлета.
Compilation – компиляция исходного кода сервлета.
Class Loading – загрузка скомпилированного класса в память.
Instantiation – внедрение конструктора без параметра загруженного класса для инициализации в памяти.
Initialization – вызов init() метода объекта JSP класса и инициализация конфигурации сервлета с первоначальными параметрами, которые указаны в дескрипторе развертывания (web.xml). После этой фазы JSP способен обрабатывать запросы клиентов. Обычно эти фазы происходят после первого запроса клиента (т.е. ленивая загрузка), но можно настроить загрузку и инициализацию JSP на старте приложения по аналогии с сервлетами.
Request Processing – длительный жизненный цикл обработки запросов клиента JSP страницей. Обработка является многопоточной и аналогична сервлетам — для каждого запроса создается новый поток, объекты ServletRequest и ServletResponse, происходит выполнение сервис методов.
Destroy – последняя фаза жизненного цикла JSP, на которой её класс удаляется из памяти. Обычно это происходит при выключении сервера или выгрузке приложения.

Расскажите о методах жизненного цикла JSP.

Контейнер сервлетов (например, Tomcat, GlassFish) создает из JSP-страницы класс сервлета, наследующего свойства интерфейса javax.servlet.jsp.HttpJspBase и включающего следующие методы:

jspInit() — метод объявлен в JSP странице и реализуется с помощью контейнера. Этот метод вызывается один раз в жизненном цикле JSP для того, чтобы инициализировать конфигурационные параметры указанные в дескрипторе развертывания. Этот метод можно переопределить с помощью определения элемента JSP scripting и указания необходимых параметров для инициализации;
_jspService() — метод переопределяется контейнером автоматически и соответствует непосредственно коду JSP, описанному на странице. Этот метод определен в интерфейсе HttpJspPage, его имя начинается с нижнего подчеркивания и он отличается от других методов жизненного цикла тем, что его невозможно переопределить;
jspDestroy() — метод вызывается контейнером для удаления объекта из памяти (на последней фазе жизненного цикла JSP - Destroy). Метод вызывается только один раз и доступен для переопределения, предоставляя возможность освободить ресурсы, которые были созданы в jspInit().

Какие методы жизненного цикла JSP могут быть переопределены?

Возможно переопределить лишь jspInit() и jspDestroy() методы.

Как можно предотвратить прямой доступ к JSP странице из браузера?

Прямой доступ к директории /WEB-INF/ из веб-приложения отсутствует. Поэтому JSP-страницы можно расположить внутри этой папки и тем самым запретить доступ к странице из браузера. Однако, по аналогии с описанием сервлетов, будет необходимо настроить дескриптор развертывания:

```
<servlet>
  <servlet-name>Example</servlet-name>
  <jsp-file>/WEB-INF/example.jsp</jsp-file>
  <init-param>
    <param-name>exampleParameter</param-name>
    <param-value>parameterValue</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>Example</servlet-name>
  <url-pattern>/example.jsp</url-pattern>
</servlet-mapping>
```

Какая разница между динамическим и статическим содержимым JSP?

Статическое содержимое JSP (HTML, код JavaScript, изображения и т.д.) не изменяется в процессе работы веб приложения.

Динамические ресурсы созданы для того, чтобы отображать свое содержимое в зависимости от пользовательских действий. Обычно они представлены в виде выражений EL (Expression Language), библиотек JSP-тегов и пр.

Как закомментировать код в JSP?

<!—- HTML комментарий; отображается на странице JSP —-> такие комментарии будут видны клиенту при просмотре кода страницы.

<%—- JSP комментарий; не отображается на странице JSP —-%> такие комментарии описываются в созданном сервлете и не посылаются клиенту. Для любых комментариев по коду или отладочной информации необходимо использовать именно такой тип комментариев.

Какие существуют основные типы тегов JSP?

Выражение JSP: <%= expression %> - выражение, которое будет обработано с перенаправлением результата на вывод;

Скриплет JSP: <% code %> - код, добавляемый в метод service().

Объявление JSP: <%! code %> - код, добавляемый в тело класса сервлета вне метода service().

Директива JSP page: <%@ page att=»value» %> - директивы для контейнера сервлетов с информацией о параметрах.

Директива JSP include: <%@ include file=»url» %> - файл в локальной системе, подключаемый при

трансляции JSP в сервлет.

Комментарий JSP: <%-- comment --%> - комментарий; игнорируется при трансляции JSP страницы в сервлет.

Что вы знаете о действиях JSP (Action tag и JSP Action Elements).

Action tag и JSP Action Elements предоставляют методы работы с Java Beans, подключения ресурсов, проброса запросов и создания динамических XML элементов. Такие элементы всегда начинаются с записи jsp: и используются непосредственно внутри страницы JSP без необходимости подключения сторонних библиотек или дополнительных настроек.

Наиболее часто используемыми JSP Action Elements являются:

jsp:include: <jsp:include page=»относительный URL» flush=»true»/> - подключить файл при запросе страницы. Если необходимо, чтобы файл подключался в процессе трансляции страницы, то используется директива page совместно с атрибутом include;

jsp:useBean: <jsp:useBean att=значение*/> или <jsp:useBean att=значение*>...</jsp:useBean> - найти или создать Java bean;

jsp:setProperty: <jsp:setProperty att=значение*/> - установить свойства Java bean, или явно, или указанием на соответствующее значение, передаваемое при запросе;

jsp:forward: <jsp:forward page=»относительный URL»/> - передать запрос другой странице;

jsp:plugin: <jsp:plugin attribute=»значение»*>...</jsp:plugin> - сгенерировать (в зависимости от типа браузера) тэги OBJECT или EMBED для апплета, использующего технологию Java Plugin.

Взаимодействие JSP - сервлет - JSP.

«JSP - сервлет - JSP» архитектура построения приложений носит название MVC (Model/View/Controller):

Model - классы данных и бизнес-логики;

View - страницы JSP;

Controller - сервлеты.

Какие области видимости переменных существуют в JSP?

Область видимости объектов определяется тем контекстом в который помещается данный объект. В зависимости от той или иной области действия так же определяется время существования объекта.

В JSP предусмотрены следующие области действия переменных (объектов):

request область действия запроса - объект будет доступен на текущей JSP странице, странице пересылки (при использовании jsp:forward) или на включаемой странице (при использовании jsp:include);

session область действия сессии - объект будет помещен в сеанс пользователя, будет доступен на всех JSP страницах и будет существовать пока существует сессия пользователя, или он не будет из нее принудительно удален.

application область действия приложения - объект будет доступен для всех пользователей на всех JSP страницах и будет существовать на протяжении всей работы приложения или пока не будет

удален принудительно и контекста приложения.

page область действия страницы - объект будет доступен только на той странице, где он определен. На включаемых (jsp:include) и переадресуемых (jsp:forward) страницах данный объект уже не будет доступен.

Таким образом, чтобы объект был доступен всем JSP страницам, необходимо указать область видимости application или session, в зависимости от того требуется ли доступ к объекту всем пользователям или только текущему.

Для указания требуемой области действия при определении объекта на JSP странице используется атрибут scope тега jsp:useBean:

```
<jsp:useBean id=»myBean» class=»ru.javacore.MyBean» scope=»session»/>
```

Если не указывать атрибут scope, то по умолчанию задается область видимости страницы page

Какие неявные, внутренние объекты и методы есть на JSP странице?

JSP implicit objects (неявные объекты) создаются контейнером при конвертации JSP страницы в код сервлета для помощи разработчикам. Эти объекты можно использовать напрямую в скриптлетах для передачи информации в сервис методы, однако мы не можем использовать неявные объекты в JSP Declaration, т.к. такой код пойдет на уровень класса.

Существует 9 видов неявных объектов, которые можно использовать прямо на JSP странице. Семь из них объявлены как локальные переменные в начале _jspService() метода, а два оставшихся могут быть использованы как аргументы метода _jspService().

```
   out Object :

<strong>Current Time is</strong>: <% out.print(new Date()); %><br>

   request Object :

<strong>Request User-Agent</strong>: <%=request.getHeader(«User-Agent») %><br>

   response Object :

<strong>Response</strong>: <%response.addCookie(new Cookie(«Test»,»Value»)); %>

   config Object :

<strong>User init param value</strong>: <%=config.getInitParameter(«User») %><br>

   application Object :

<strong>User context param value</strong>: <%=application.getInitParameter(«User») %><br>

   session Object :

<strong>User Session ID</strong>: <%=session.getId() %><br>

   pageContext Object :
```

```
<% pageContext.setAttribute(«Test», «Test Value»); %>
<strong>PageContext attribute</strong>: {Name=»Test»,Value=»<%=pageContext.
getAttribute(«Test») %>»}<br>

   page Object :

<strong>Generated Servlet Name</strong>: <%=page.getClass().getName() %>

   exception Object :
<strong>Exception occured</strong>: <%=exception %><br>
```

Какие неявные объекты не доступны в обычной JSP странице?

Неявный объект исключений JSP недоступен в обычных JSP страницах и используется на страницах ошибок JSP (errorpage) только для того, чтобы перехватить исключение, выброшенное JSP страницей и далее предоставить какую-либо полезную информацию клиенту.

Что вы знаете о PageContext и какие преимущества его использования?

Неявный объект JSP - экземпляр класса javax.servlet.jsp.PageContext предоставляет доступ ко всем пространствам имён, ассоциированным с JSP-страницей, а также к различным её атрибутам. Остальные неявные объекты добавляются к pageContext автоматически.

Класс PageContext это абстрактный класс, а его экземпляр можно получить через вызов метода JspFactory.getPageContext(), и освободить через вызов метода JspFactory.releasePageContext().

PageContext обладает следующим набором особенностей и возможностей:

- единый API для обслуживания пространств имён различных областей видимости;
- несколько удобных API для доступа к различным public-объектам;
- механизм получения JspWriter для вывода;
- механизм обслуживания использования сессии страницей;
- механизм экспонирования («показа») атрибутов директивы page среде скриптинга;
- механизмы направления или включения текущего запроса в другие компоненты приложения;
- механизм обработки процессов исключений на страницах ошибок errorpage;

Как сконфигурировать параметры инициализации для JSP?

Параметры инициализации для JSP задаются в web.xml файле аналогично сервлетам - элементами servlet и servlet-mapping. Единственным отличием будет указание местонахождения JSP страницы:

```
<servlet>
  <servlet-name>Example</servlet-name>
  <jsp-file>/WEB-INF/example.jsp</jsp-file>
  <init-param>
    <param-name>exampleParameter</param-name>
    <param-value>parameterValue</param-value>
  </init-param>
</servlet>
```

Почему не рекомендуется использовать скриплеты (скриптовые элементы) в JSP?

JSP страницы используются в основном для целей отображения представления (view), а вся бизнес-логика (controller) и модель (model) должны быть реализованы в сервлетах или классах-моделях. Обязанность JSP страницы - создание HTML ответа из переданных через атрибуты параметров. Большая часть JSP содержит HTML код а для того, чтобы помочь верстальщикам понять JSP код страницы предоставляется функционал элементов action, JSP EL, JSP Standart Tag Library. Именно их и необходимо использовать вместо скриптлетов для создания моста между (JSP)HTML и (JSP) Java частями.

Можно ли определить класс внутри JSP страницы?

Определить класс внутри JSP страницы можно, но это считается плохой практикой:

```
<%!
private static class ExampleOne {
  //...
}
%>

<%
private class ExampleTwo {
  //...
}
%>
```

Что вы знаете о Языке выражений JSP (JSP Expression Language – EL)?

JSP Expression Language (EL) — скриптовый язык выражений, который позволяет получить доступ к Java компонентам (JavaBeans) из JSP. Начиная с JSP 2.0 используется внутри JSP тегов для отделения Java кода от JSP для обеспечения лёгкого доступа к Java компонентам, уменьшая при этом количество кода Java в JSP-страницах, или даже полностью исключая его.

Развитие EL происходило с целью сделать его более простым для дизайнеров, которые имеют минимальные познания в языке программирования Java. До появления языка выражений, JSP имел несколько специальных тегов таких как скриптлеты (англ.), выражения и т. п. которые позволяли записывать Java код непосредственно на странице. С использованием языка выражений веб-дизайнер должен знать только то, как организовать вызов соответствующих java-методов.

Язык выражений JSP 2.0 включает:

- Создание и изменение переменных.
- Управление потоком выполнения программы: ветвление, выполнение различных типов итераций и т.д.
- Упрощенное обращение к встроенным JSP-объектам.
- Возможность создавать собственные функции.

Язык выражений используется внутри конструкции ${ ... }. Подобная конструкция может размещаться либо отдельно, либо в правой части выражения установки атрибута тега.

Какие типы EL операторов вы знаете?

Операторы в EL поддерживают наиболее часто используемые манипуляции данными.

Типы операторов:

Стандартные операторы отношения: == (или eq), != (или neq), < (или lt), > (или gt), <= (или le), >= (или ge).
Арифметические операторы: +, –, *, / (или div), % (или mod).
Логические операторы: && (или and), || (или or), ! (или not).
Оператор empty – используется для проверки переменной на null, или «пустое значение», который зависит от типа проверяемого объекта. Например, нулевая длина для строки или нулевой размер для коллекции.

Назовите неявные, внутренние объекты JSP EL и их отличия от объектов JSP.

Язык выражений JSP предоставляет множество неявных объектов, которые можно использовать для получения атрибутов в различных областях видимости (scopes) и для значений параметров. Важно отметить, что они отличаются от неявных объектов JSP и содержат атрибуты в заданной области видимости. Наиболее часто использующийся implicit object в JSP EL и JSP page — это объект pageContext. Ниже представлена таблица неявных объектов JSP EL.

Как отключить возможность использования EL в JSP?

Для игнорирования выполнения языка выражений на странице существует два способа:

использовать директиву <%@ page isELIgnored = «true» %>,
настроить web.xml (лучше подходит для отключения EL сразу на нескольких страницах):

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>
</jsp-config>
```

Как узнать тип HTTP метода используя JSP EL?

${pageContext.request.method}.

Что такое JSTL (JSP Standard tag library)?

JavaServer Pages Standard Tag Library, JSTL, Стандартная библиотека тегов JSP — расширение спецификации JSP (конечный результат JSR 52), добавляющее библиотеку JSP тегов для общих нужд, таких как разбор XML данных, условная обработка, создание циклов и поддержка интернационализации.

JSTL является альтернативой такому виду встроенной в JSP логики, как скриплеты (прямые вставки Java кода). Использование стандартизованного множества тегов предпочтительнее, поскольку получаемый код легче поддерживать и проще отделять бизнес-логику от логики отображения.

Для использования JSTL тегов необходимо:

- подключить зависимости, например в pom.xml:

```
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>1.1.2</version>
</dependency>
```

- указать пространство имен основных тегов JSTL через указание на JSP странице код:

```
<%@ taglib uri=»http://java.sun.com/jsp/jstl/core» prefix=»c» %>
<%@ taglib uri=»http://java.sun.com/jsp/jstl/fmt» prefix=»fmt» %>
<%@ taglib uri=»http://java.sun.com/jsp/jstl/sql» prefix=»sql» %>
<%@ taglib uri=»http://java.sun.com/jsp/jstl/xml» prefix=»x» %>
<%@ taglib uri=»http://java.sun.com/jsp/jstl/functions» prefix=»fn» %>
```

Из каких групп тегов состоит библиотека JSTL?

Группы тегов JSTL согласно их функциональности:

Core Tags предоставляют возможности итерации, обработки исключений, URL, forward, redirect response и т.д.

Formatting Tags и Localization Tags предоставляют возможности по форматированию чисел, дат и поддержки i18n локализации и resource bundles.

SQL Tags – поддержка работы с базами данных.

XML Tags используются для работы с XML документами: парсинга, преобразования данных, выполнения выражений XPath и т.д..

JSTL Functions Tags предоставляет набор функций, которые позволяют выполнять различные операции со строками и т.п. Например, по конкатенации или разбиению строк.

Какая разница между <c:set> и <jsp:useBean>?

Оба тега создают и помещают экземпляры в заданную область видимости, но <jsp:useBean> только создаёт экземпляр конкретного типа, а <c:set>, создав экземпляр, позволяет дополнительно извлекать значение, например, из параметров запроса, сессии и т. д.

Чем отличается <c:import> от <jsp:include> и директивы <%@include %>?

По сравнению с action-тегом <jsp:include> и директивой <%@include %> тег <c:import> обеспечивает более совершенное включение динамических ресурсов, т.к. получает доступ к источнику, чтение информации из которого происходит непосредственно без буферизации и контент включается в исходную JSP построчно.

Как можно расширить функциональность JSP? Что вы знаете о написании пользовательских JSP тегов?

Приведите пример использования собственных тегов.

JSP можно расширить с помощью создания собственных тегов с необходимой функциональностью, которые можно добавить в библиотеку тегов на страницу JSP указав необходимое пространство имен.

/WEB-INF/exampleTag.tld

```
<?xml version=»1.0» encoding=»UTF-8»?>
<taglib version=»2.1» xmlns=»http://java.sun.com/xml/ns/j2ee»
  xmlns:xsi=»http://www.w3.org/2001/XMLSchema-instance»
  xsi:schemaLocation=»http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_1.xsd»>
  <tlib-version>1.0</tlib-version>
  <short-name>example</short-name>
  <uri>/WEB-INF/exampleTag</uri>
  <tag>
    <name>exampleTag</name>
    <tag-class>xyz.company.ExampleTag</tag-class>
    <body-content>empty</body-content>
    <info>The example tag displays Hello World!</info>
  </tag>
</taglib>
```

xyz.company.ExampleServlet.java

```
package xyz.company;

import java.io.IOException;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

public class ExampleTag extends TagSupport{
  private static final long serialVersionUID = 1L;

  @Override
  public int doStartTag() throws JspException {
    try {
      pageContext.getOut().print(«Hello World!»);
    } catch(IOException ioException) {
      throw new JspException(«Error: « + ioException.getMessage());
    }
    return SKIP_BODY;
  }
}
```

exampleTag.jsp

```
<%@ taglib uri=»/WEB-INF/exampleTag.tld» prefix=»example»%>
<%@ page session=»false» pageEncoding=»UTF-8»%>
<html>
<head>
<title>Example Tag</title>
</head>
```

```
<body>
  <h1>Example Tag</h1>
  <p><example:exampleTage /><p>
</body>
</html>
```

Также в пользовательских тегах существует возможность задать входные параметры. Например, существует необходимость отформатировать каким-либо стилем очень длинное число. Для этого можно использовать собственный тег по типу:

```
<mytags:formatNumber number=»123456.789» format=»#,## #.00»/>
```

Используя входные параметры, число должно быть преобразовано на JSP странице в таком виде 123,456.79 согласно шаблону. Т.к. JSTL не предоставляет такой функциональности, необходимо создать пользовательский тег для получения необходимого результата.

Как сделать перенос строки в HTML средствами JSP?

Для переноса строки можно использовать тег c:out и атрибут escapeXml, который отключает обработку HTML элементов. В этом случае браузер получит следующий код в виде строки и обработает элемент
 как требуется:

```
<c:out value=»<br> creates a new line in HTML» escapeXml=»true»></c:out>
```

Почему не нужно конфигурировать стандартные JSP теги в web.xml?

Стандартные теги JSP не конфигурируются в web.xml, потому что tld файлы уже находятся внутри каталога /META-INF/ в jar файлах JSTL.

Когда контейнер загружает веб-приложение и находит tld файлы в в jar файле в директории /META-INF/, то он автоматически настраивает их для непосредственного использования на JSP страницах. Остается только задать пространство имен на JSP странице.

Как можно обработать ошибки JSP страниц?

Для обработки исключений выброшенных на JSP странице достаточно лишь задать страницу ошибки JSP и при её создании установить значение page directive attribute isErrorPage в значение true. Таким образом будет предоставлен доступ к неявным объектам исключений в JSP и появится возможность передавать собственные, более информативные сообщения об ошибках клиенту. При этом настройка дескриптора развертывания выглядит так:

```
<error-page>
   <error-code>404</error-code>
   <location>/error.jsp</location>
</error-page>

<error-page>
   <exception-type>java.lang.Throwable</exception-type>
   <location>/error.jsp</location>
</error-page>
```

Как происходит обработка ошибок с помощью JSTL?

Для перехвата и обработки исключений в служебных методах класса служат JSTL Core Tags c:catch и c:if.

Тег c:catch перехватывает исключение и обертывает его в переменную exception, доступную для обработки в теге c:if:

```
<c:catch var =»exception»>
  <% int x = 42/0;%>
</c:catch>
<c:if test = «${exception ne null}»>
  <p>Exception is : ${exception} <br />
  Exception Message: ${exception.message}</p>
</c:if>
```

Обратите внимание что используется язык выражений JSP EL в теге c:if.

Как конфигурируется JSP в дескрипторе развертывания.

Для настройки различных параметров JSP страниц используется элемент jsp-config, который отвечает за:

- управление элементами скриптлетов на странице;
- управления выполнением в языке выражений;
- определение шаблона URL для encoding;
- определение размера буфера, который используется для объектов на странице;
- обозначение групп ресурсов, соответствующих шаблону URL, которые должны быть обработаны как XML документ.

```
<jsp-config>
  <taglib>
    <taglib-uri>http://company.xyz/jsp/tlds/customtags</taglib-uri>
    <taglib-location>/WEB-INF/exampleTag.tld</taglib-location>
  </taglib>
</jsp-config>
```

Можно ли использовать Javascript на JSP странице?

Да, это возможно. Несмотря на то, что JSP это серверная технология, на выходе она всё равно создает HTML страницу, на которую можно добавлять Javascript и CSS.

Всегда ли создается объект сессии на JSP странице, можно ли отключить его создание?

Jsp-страница, по умолчанию, всегда создает сессию. Используя директиву page с атрибутом session можно изменить это поведение:

```
<%@ page session =»false» %>
```

Какая разница между JSPWriter и сервлетным PrintWriter?

PrintWriter является объектом отвечающим за запись содержания ответа на запрос. JspWriter использует объект PrintWriter для буферизации. Когда буфер заполняется или сбрасывается, JspWriter использует объект PrintWriter для записи содержания в ответ.

Опишите общие практические принципы работы с JSP.

Хорошей практикой работы с технологией JSP является соблюдение следующих правил:

Следует избегать использования элементов скриптлетов на странице. Если элементы action, JSTL, JSP EL не удовлетворяют потребностям, то желательно написать собственный тег.

Рекомендуется использовать разные виды комментариев: так JSP комментарии необходимы для уровня кода и отладки, т.к. они не будут показаны клиенту.

Не стоит размещать какой-либо бизнес логики внутри JSP страницы. Страницы должны использоваться только для создания ответов клиенту.

Для повышения производительности лучше отключать создание сессии на странице, когда это не требуется.

Директивы taglib, page в начале JSP страницы улучшают читабельность кода.

Следует правильно использовать директиву include и элемент jsp:include action. Первая используется для статических ресурсов, а второй для динамических ресурсов времени выполнения.

Обработку исключений нужно производить с помощью страниц ошибок. Это помогает избегать запуска специальных служебных методов и может повысить производительность.

Использующиеся CSS и JavaScript должны быть разнесены в разные файлы и подключаться в начале страницы.

В большинстве случаев JSTL должно хватать для всех нужд. Если это не так, то в начале следует проанализировать логику своего приложения, и попробовать перенести выполнения кода в сервлет, а далее с помощью установки атрибутов использовать на JSP странице только результат.

Базы данных

Что такое «база данных»?

База данных — организованный и адаптированный для обработки вычислительной системой набор информации.

Что такое «система управления базами данных»?

Система управления базами данных (СУБД) - набор средств общего или специального назначения, обеспечивающий создание, доступ к материалам и управление базой данных.

Основные функции СУБД:

- управление данными
- журнализация изменений данных
- резервное копирование и восстановление данных;
- поддержка языка определения данных и манипулирования ими.

Что такое «реляционная модель данных»?

Реляционная модель данных — это логическая модель данных и прикладная теория построения реляционных баз данных.

Реляционная модель данных включает в себя следующие компоненты:

Структурный аспект — данные представляют собой набор отношений.
Аспект целостности — отношения отвечают определенным условиям целостности: уровня домена (типа данных), уровня отношения и уровня базы данных.
Аспект обработки (манипулирования) — поддержка операторов манипулирования отношениями (реляционная алгебра, реляционное исчисление).
Нормальная форма - свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности и определённое как совокупность требований, которым должно удовлетворять отношение.

Дайте определение терминам «простой», «составной» (composite), «потенциальный» (candidate) и «альтернативный» (alternate) ключ.

Простой ключ состоит из одного атрибута (поля). Составной - из двух и более.

Потенциальный ключ - простой или составной ключ, который уникально идентифицирует каждую запись набора данных. При этом потенциальный ключ должен обладать критерием неизбыточности: при удалении любого из полей набор полей перестает уникально идентифицировать запись.

Из множества всех потенциальных ключей набора данных выбирают первичный ключ, все остальные ключи называют альтернативными.

Что такое «первичный ключ» (primary key)? Каковы критерии его выбора?

Первичный ключ (primary key) в реляционной модели данных один из потенциальных ключей отношения, выбранный в качестве основного ключа (ключа по умолчанию).
Если в отношении имеется единственный потенциальный ключ, он является и первичным ключом.

Если потенциальных ключей несколько, один из них выбирается в качестве первичного, а другие называют «альтернативными».

В качестве первичного обычно выбирается тот из потенциальных ключей, который наиболее удобен. Поэтому в качестве первичного ключа, как правило, выбирают тот, который имеет наименьший размер (физического хранения) и/или включает наименьшее количество атрибутов. Другой критерий выбора первичного ключа — сохранение его уникальности со временем. Поэтому в качестве первичного ключа стараются выбирать такой потенциальный ключ, который с наибольшей вероятностью никогда не утратит уникальность.

Что такое «внешний ключ» (foreign key)?

Внешний ключ (foreign key) — подмножество атрибутов некоторого отношения A, значения которых должны совпадать со значениями некоторого потенциального ключа некоторого отношения B.

Что такое «нормализация»?

Нормализация - это процесс преобразования отношений базы данных к виду, отвечающему нормальным формам (пошаговый, обратимый процесс замены исходной схемы другой схемой, в которой наборы данных имеют более простую и логичную структуру).

Нормализация предназначена для приведения структуры базы данных к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объёма базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации.

Какие существуют нормальные формы?

Первая нормальная форма (1NF) - Отношение находится в 1NF, если значения всех его атрибутов атомарны (неделимы).

Вторая нормальная форма (2NF) - Отношение находится в 2NF, если оно находится в 1NF, и при этом все неключевые атрибуты зависят только от ключа целиком, а не от какой-то его части.

Третья нормальная форма (3NF) - Отношение находится в 3NF, если оно находится в 2NF и все неключевые атрибуты не зависят друг от друга.

Четвёртая нормальная форма (4NF) - Отношение находится в 4NF , если оно находится в 3NF и если в нем не содержатся независимые группы атрибутов, между которыми существует отношение «многие-ко-многим».

Пятая нормальная форма (5NF) - Отношение находится в 5NF, когда каждая нетривиальная зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения.

Шестая нормальная форма (6NF) - Отношение находится в 6NF, когда она удовлетворяет всем нетривиальным зависимостям соединения, т.е. когда она неприводима, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Каждая переменная отношения, которая находится в 6NF, также находится и в 5NF. Введена как обобщение пятой нормальной формы для хронологической базы данных.

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

Доменно-ключевая нормальная форма (DKNF) - Отношение находится в DKNF, когда каждое наложенное на неё ограничение является логическим следствием ограничений доменов и ограничений ключей, наложенных на данное отношение.

Что такое «денормализация»? Для чего она применяется?

Денормализация базы данных — это процесс осознанного приведения базы данных к виду, в котором она не будет соответствовать правилам нормализации. Обычно это необходимо для повышения производительности и скорости извлечения данных, за счет увеличения избыточности данных.

Какие существуют типы связей в базе данных? Приведите примеры.

Один к одному - любому значению атрибута A соответствует только одно значение атрибута B, и наоборот.

Каждый университет гарантированно имеет 1-го ректора: 1 университет → 1 ректор.

Один ко многим - любому значению атрибута A соответствует 0, 1 или несколько значений атрибута B.

В каждом университете есть несколько факультетов: 1 университет → много факультетов.

Многие ко многим - любому значению атрибута A соответствует 0, 1 или несколько значений атрибута B, и любому значению атрибута B соответствует 0, 1 или несколько значение атрибута A.

1 профессор может преподавать на нескольких факультетах, в то же время на 1-ом факультете может преподавать несколько профессоров: Несколько профессоров ↔Несколько факультетов.

Что такое «индексы»? Для чего их используют? В чём заключаются их преимущества и недостатки?

Индекс (index) — объект базы данных, создаваемый с целью повышения производительности выборки данных.

Наборы данных могут иметь большое количество записей, которые хранятся в произвольном порядке, и их поиск по заданному критерию путём последовательного просмотра набора данных запись за записью может занимать много времени. Индекс формируется из значений одного или нескольких полей и указателей на соответствующие записи набора данных, - таким образом, достигается значительный прирост скорости выборки из этих данных.

Преимущества

- ускорение поиска и сортировки по определенному полю или набору полей.
- обеспечение уникальности данных.

Недостатки

- требование дополнительного места на диске и в оперативной памяти и чем больше/длиннее ключ, тем больше размер индекса.
- замедление операций вставки, обновления и удаления записей, поскольку при этом приходится обновлять сами индексы.

Индексы предпочтительней для:

- Поля-счетчика, чтобы в том числе избежать и повторения значений в этом поле;
- Поля, по которому проводится сортировка данных;
- Полей, по которым часто проводится соединение наборов данных. Поскольку в этом случае данные располагаются в порядке возрастания индекса и соединение происходит значительно быстрее;
- Поля, которое объявлено первичным ключом (primary key);
- Поля, в котором данные выбираются из некоторого диапазона. В этом случае как только будет найдена первая запись с нужным значением, все последующие значения будут расположены рядом.

Использование индексов нецелесообразно для:

- Полей, которые редко используются в запросах;
- Полей, которые содержат всего два или три значения, например: мужской, женский пол или значения «да», «нет».

Какие типы индексов существуют?

По порядку сортировки

- упорядоченные — индексы, в которых элементы упорядочены;
- возрастающие;
- убывающие;
- неупорядоченные — индексы, в которых элементы неупорядочены.

По источнику данных

- индексы по представлению (view);
- индексы по выражениям.

По воздействию на источник данных

- кластерный индекс - при определении в наборе данных физическое расположение данных перестраивается в соответствии со структурой индекса. Логическая структура набора данных в этом случае представляет собой скорее словарь, чем индекс. Данные в словаре физически упорядочены, например по алфавиту. Кластерные индексы могут дать существенное увеличение производительности поиска данных даже по сравнению с обычными индексами. Увеличение производительности особенно заметно при работе с последовательными данными.
- некластерный индекс — наиболее типичные представители семейства индексов. В отличие от кластерных, они не перестраивают физическую структуру набора данных, а лишь организуют ссылки на соответствующие записи. Для идентификации нужной записи в наборе данных некластерный индекс организует специальные указатели, включающие в себя: информацию об идентификационном номере файла, в котором хранится запись; идентификационный номер страницы соответствующих данных; номер искомой записи на соответствующей странице; содержимое столбца.

По структуре

B*-деревья;
B+-деревья;
B-деревья;
Хэши.

По количественному составу
- простой индекс (индекс с одним ключом) — строится по одному полю;

- составной (многоключевой, композитный) индекс — строится по нескольким полям при этом важен порядок их следования;
- индекс с включенными столбцами — некластеризованный индекс, дополнительно содержащий кроме ключевых столбцов еще и неключевые;
- главный индекс (индекс по первичному ключу) — это тот индексный ключ, под управлением которого в данный момент находится набор данных. Набор данных не может быть отсортирован по нескольким индексным ключам одновременно. Хотя, если один и тот же набор данных открыт одновременно в нескольких рабочих областях, то у каждой копии набора данных может быть назначен свой главный индекс.

По характеристике содержимого

- уникальный индекс состоит из множества уникальных значений поля;
- плотный индекс (NoSQL) — индекс, при котором, каждом документе в индексируемой коллекции соответствует запись в индексе, даже если в документе нет индексируемого поля.
- разреженный индекс (NoSQL) — тот, в котором представлены только те документы, для которых индексируемый ключ имеет какое-то определённое значение (существует).
- пространственный индекс — оптимизирован для описания географического местоположения. Представляет из себя многоключевой индекс состоящий из широты и долготы.
- составной пространственный индекс — индекс, включающий в себя кроме широты и долготы ещё какие-либо мета-данные (например теги). Но географические координаты должны стоять на первом месте.
- полнотекстовый (инвертированный) индекс — словарь, в котором перечислены все слова и указано, в каких местах они встречаются. При наличии такого индекса достаточно осуществить поиск нужных слов в нём и тогда сразу же будет получен список документов, в которых они встречаются.
- хэш-индекс предполагает хранение не самих значений, а их хэшей, благодаря чему уменьшается размер (а, соответственно, и увеличивается скорость их обработки) индексов из больших полей. Таким образом, при запросах с использованием хэш-индексов, сравниваться будут не искомое со значения поля, а хэш от искомого значения с хэшами полей. Из-за нелинейнойсти хэш-функций данный индекс нельзя сортировать по значению, что приводит к невозможности использования в сравнениях больше/меньше и «is null». Кроме того, так как хэши не уникальны, то для совпадающих хэшей применяются методы разрешения коллизий.
- битовый индекс (bitmap index) — метод битовых индексов заключается в создании отдельных битовых карт (последовательностей 0 и 1) для каждого возможного значения столбца, где каждому биту соответствует запись с индексируемым значением, а его значение равное 1 означает, что запись, соответствующая позиции бита содержит индексируемое значение для данного столбца или свойства.
- обратный индекс (reverse index) — B-tree индекс, но с реверсированным ключом, используемый в основном для монотонно возрастающих значений (например, автоинкрементный идентификатор) в OLTP системах с целью снятия конкуренции за последний листовой блок индекса, т.к. благодаря переворачиванию значения две соседние записи индекса попадают в разные блоки индекса. Он не может использоваться для диапазонного поиска.
- функциональный индекс, индекс по вычисляемому полю (function-based index) — индекс, ключи которого хранят результат пользовательских функций. Функциональные индексы часто строятся для полей, значения которых проходят предварительную обработку перед сравнением в команде SQL. Например, при сравнении строковых данных без учета регистра символов часто используется функция UPPER. Кроме того, функциональный индекс может помочь реализовать любой другой отсутствующий тип индексов данной СУБД.
- первичный индекс — уникальный индекс по полю первичного ключа.
- вторичный индекс — индекс по другим полям (кроме поля первичного ключа).
- XML-индекс — вырезанное материализованное представление больших двоичных XML-объектов (BLOB) в столбце с типом данных xml.

По механизму обновления

- полностью перестраиваемый — при добавлении элемента заново перестраивается весь индекс.
- пополняемый (балансируемый) — при добавлении элементов индекс перестраивается частично (например одна из ветви) и периодически балансируется.

По покрытию индексируемого содержимого

- полностью покрывающий (полный) индекс — покрывает всё содержимое индексируемого объекта.
- частичный индекс (partial index) — это индекс, построенный на части набора данных, удовлетворяющей определенному условию самого индекса. Данный индекс создан для уменьшения размера индекса.
- инкрементный (delta) индекс — индексируется малая часть данных(дельта), как правило, по истечении определённого времени. Используется при интенсивной записи. Например, полный индекс перестраивается раз в сутки, а дельта-индекс строится каждый час. По сути это частичный индекс по временной метке.
- индекс реального времени (real-time index) — особый вид инкрементного индекса, характеризующийся высокой скоростью построения. Предназначен для часто меняющихся данных.

Индексы в кластерных системах

- глобальный индекс — индекс по всему содержимому всех сегментов БД (shard).
- сегментный индекс — глобальный индекс по полю-сегментируемому ключу (shard key). Используется для быстрого определения сегмента, на котором хранятся данные в процессе маршрутизации запроса в кластере БД.
- локальный индекс — индекс по содержимому только одного сегмента БД.

В чем отличие между кластерными и некластерными индексами?

Некластерные индексы - данные физически расположены в произвольном порядке, но логически упорядочены согласно индексу. Такой тип индексов подходит для часто изменяемого набора данных.

При кластерном индексировании данные физически упорядочены, что серьезно повышает скорость выборок данных (но только в случае последовательного доступа к данным). Для одного набора данных может быть создан только один кластерный индекс.

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. Также, 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 зароса и/или слабой кластеризации данных, либо в случае очень маленьких наборов данных.

Что такое «транзакция»?

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

Назовите основные свойства транзакции.

Атомарность (atomicity) гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной.

Согласованность (consistency). Транзакция, достигающая своего нормального завершения и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных.

Изолированность (isolation). Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат.

Долговечность (durability). Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу.

Какие существуют уровни изолированности транзакций?

В порядке увеличения изолированности транзакций и, соответственно, надёжности работы с данными:

- Чтение неподтверждённых данных (грязное чтение) (read uncommitted, dirty read) — чтение незафиксированных изменений как своей транзакции, так и параллельных транзакций. Нет гарантии, что данные, изменённые другими транзакциями, не будут в любой момент изменены в результате их отката, поэтому такое чтение является потенциальным источником ошибок. Невозможны потерянные изменения, возможны неповторяемое чтение и фантомы.
- Чтение подтверждённых данных (read committed) — чтение всех изменений своей транзакции и зафиксированных изменений параллельных транзакций. Потерянные изменения и грязное чтение не допускается, возможны неповторяемое чтение и фантомы.
- Повторяемость чтения (repeatable read, snapshot) — чтение всех изменений своей транзакции, любые изменения, внесённые параллельными транзакциями после начала своей, недоступны. Потерянные изменения, грязное и неповторяемое чтение невозможны, возможны фантомы.
- Упорядочиваемость (serializable) — результат параллельного выполнения сериализуемой транзакции с другими транзакциями должен быть логически эквивалентен результату их какого-либо последовательного выполнения. Проблемы синхронизации не возникают.

Какие проблемы могут возникать при параллельном доступе с использованием транзакций?

При параллельном выполнении транзакций возможны следующие проблемы:

- Потерянное обновление (lost update) — при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется;
- «Грязное» чтение (dirty read) — чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится);
- Неповторяющееся чтение (non-repeatable read) — при повторном чтении в рамках одной тран-

закции ранее прочитанные данные оказываются изменёнными;

- Фантомное чтение (phantom reads) — одна транзакция в ходе своего выполнения несколько раз выбирает множество записей по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет или удаляет записи или изменяет столбцы некоторых записей, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества записей.

SQL

Что такое «SQL»?

SQL, Structured query language («язык структурированных запросов») — формальный непроцедурный язык программирования, применяемый для создания, модификации и управления данными в произвольной реляционной базе данных, управляемой соответствующей системой управления базами данных (СУБД).

Какие существуют операторы SQL?

операторы определения данных (Data Definition Language, DDL):

CREATE создает объект БД (базу, таблицу, представление, пользователя и т. д.),
ALTER изменяет объект,
DROP удаляет объект;

операторы манипуляции данными (Data Manipulation Language, DML):

SELECT выбирает данные, удовлетворяющие заданным условиям,
INSERT добавляет новые данные,
UPDATE изменяет существующие данные,
DELETE удаляет данные;

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;

операторы управления транзакциями (Transaction Control Language, TCL):

COMMIT применяет транзакцию,
ROLLBACK откатывает все изменения, сделанные в контексте текущей транзакции,
SAVEPOINT разбивает транзакцию на более мелкие.

Что означает NULL в SQL?

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!

Что такое «временная таблица»? Для чего она используется?

Временная таблица - это объект базы данных, который хранится и управляется системой базы данных на временной основе. Они могут быть локальными или глобальными. Используется для сохранения результатов вызова хранимой процедуры, уменьшение числа строк при соединениях,

агрегирование данных из различных источников или как замена курсоров и параметризованных представлений.

Что такое «представление» (view) и для чего оно применяется?

Представление, 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 также 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.

Каков общий синтаксис оператора SELECT?

SELECT - оператор DML SQL, возвращающий набор данных (выборку) из базы данных, удовлетворяющих заданному условию. Имеет следующую структуру:

```
SELECT
      [DISTINCT | DISTINCTROW | ALL]
      select_expression,...
  FROM table_references
    [WHERE where_definition]
    [GROUP BY {unsigned_integer | column | formula}]
    [HAVING where_definition]
    [ORDER BY {unsigned_integer | column | formula} [ASC | DESC], ...]
```

Что такое JOIN?

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 LEFT 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?

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.

Для чего используется оператор HAVING?

HAVING используется для фильтрации результата GROUP BY по заданным логическим условиям.

В чем различие между операторами HAVING и WHERE?

HAVING используется как WHERE, но в другой части SQL-выражения и, соответственно, на другой стадии формирования ответа.

Для чего используется оператор 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.

Для чего используется оператор GROUP BY?

GROUP BY используется для агрегации записей результата по заданным признакам-атрибутам.

Как GROUP BY обрабатывает значение NULL?

При использовании GROUP BY все значения NULL считаются равными.

В чем разница между операторами GROUP BY и DISTINCT?

DISTINCT указывает, что для вычислений используются только уникальные значения столбца. NULL считается как отдельное значение. GROUP BY создает отдельную группу для всех возможных значений (включая значение NULL).

Если нужно удалить только дубликаты лучше использовать DISTINCT, GROUP BY лучше использовать для определения групп записей, к которым могут применяться агрегатные функции.

Перечислите основные агрегатные функции.

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

SQL предоставляет несколько агрегатных функций:

COUNT - производит подсчет записей, удовлетворяющих условию запроса; 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.

В чем разница между COUNT(*) и COUNT({column})?

COUNT (*) подсчитывает количество записей в таблице, не игнорируя значение NULL, поскольку эта функция оперирует записями, а не столбцами.

COUNT ({column}) подсчитывает количество значений в {column}. При подсчете количества значений столбца эта форма функции COUNT не принимает во внимание значение NULL.

Что делает оператор EXISTS?

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

Для чего используются 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';

Для чего применяется ключевое слово 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.

Какие ограничения на целостность данных существуют в SQL?

PRIMARY KEY - набор полей (1 или более), значения которых образуют уникальную комбинацию и используются для однозначной идентификации записи в таблице. Для таблицы может быть создано только одно такое ограничение. Данное ограничение используется для обеспечения целостности сущности, которая описана таблицей.

CHECK используется для ограничения множества значений, которые могут быть помещены в данный столбец. Это ограничение используется для обеспечения целостности предметной области, которую описывают таблицы в базе.

UNIQUE обеспечивает отсутствие дубликатов в столбце или наборе столбцов.

FOREIGN KEY защищает от действий, которые могут нарушить связи между таблицами. FOREIGN KEY в одной таблице указывает на PRIMARY KEY в другой. Поэтому данное ограничение нацелено на то, чтобы не было записей FOREIGN KEY, которым не отвечают записи PRIMARY KEY.

Какие отличия между ограничениями PRIMARY и UNIQUE?

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

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

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

Как создать индекс?

Индекс можно создать либо с помощью выражения CREATE INDEX:
CREATE INDEX index_name ON table_name (column_name)

либо указав ограничение целостности в виде уникального UNIQUE или первичного PRIMARY клю-

ча в операторе создания таблицы CREATE TABLE.

Что делает оператор MERGE?

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.

В чем отличие между операторами DELETE и TRUNCATE?

DELETE - оператор DML, удаляет записи из таблицы, которые удовлетворяют критерию WHERE при этом задействуются триггеры, ограничения и т.д.

TRUNCATE - DDL оператор (удаляет таблицу и создает ее заново. Причем если на эту таблицу есть ссылки FOREGIN KEY или таблица используется в репликации, то пересоздать такую таблицу не получится).

Что такое «хранимая процедура»?

Хранимая процедура — объект базы данных, представляющий собой набор SQL-инструкций, который хранится на сервере. Хранимые процедуры очень похожи на обыкновенные процедуры 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 её обработка осуществляется быстрее.

Что такое «триггер»?

Триггер (trigger) — это хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением, удалением или изменением данных в заданной таблице реляционной базы данных. Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается сервером автоматически и все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Момент запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанного с ним события) или AFTER (после события). В случае, если триггер вызывается до события, он может внести изменения в модифицируемую событием запись. Кроме того, триггеры могут быть привязаны не к таблице, а к представлению (VIEW). В этом случае с их помощью реализуется механизм «обновляемого представления». В этом случае ключевые слова BEFORE и AFTER влияют лишь на последовательность вызова триггеров, так как собственно событие (удаление, вставка или обновление) не происходит.

Что такое «курсор»?

Курсор — это объект базы данных, который позволяет приложениям работать с записями «по-одной», а не сразу с множеством, как это делается в обычных SQL командах.

Порядок работы с курсором такой:
Определить курсор (DECLARE)
Открыть курсор (OPEN)
Получить запись из курсора (FETCH)
Обработать запись...
Закрыть курсор (CLOSE)
Удалить ссылку курсора (DEALLOCATE). Когда удаляется последняя ссылка курсора, SQL освобождает структуры данных, составляющие курсор.

Опишите разницу типов данных DATETIME и 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 усреднённому времени Гринвича. При получении из базы отображается с учётом часового пояса. Размер: 4 байта

Для каких числовых типов недопустимо использовать операции сложения/вычитания?

В качестве операндов операций сложения и вычитания нельзя использовать числовой тип BIT.

Какое назначение у операторов PIVOT и UNPIVOT в Transact-SQL?

PIVOT и UNPIVOT являются нестандартными реляционными операторами, которые поддерживаются Transact-SQL.

Оператор PIVOT разворачивает возвращающее табличное значение выражение, преобразуя уникальные значения одного столбца выражения в несколько выходных столбцов, а также, в случае необходимости, объединяет оставшиеся повторяющиеся значения столбца и отображает их в выходных данных. Оператор UNPIVOT производит действия, обратные PIVOT, преобразуя столбцы возвращающего табличное значение выражения в значения столбца.

Расскажите об основных функциях ранжирования в Transact-SQL.

Ранжирующие функции - это функции, которые возвращают значение для каждой записи группы в результирующем наборе данных. На практике они могут быть использованы, например, для простой нумерации списка, составления рейтинга или постраничной навигации.

ROW_NUMBER – функция нумерации в Transact-SQL, которая возвращает просто номер записи.

RANK возвращает ранг каждой записи. В данном случае, в отличие от ROW_NUMBER, идет уже анализ значений и в случае нахождения одинаковых возвращает одинаковый ранг с пропуском

следующего.

DENSE_RANK так же возвращает ранг каждой записи, но в отличие от RANK в случае нахождения одинаковых значений возвращает ранг без пропуска следующего.

NTILE – функция Transact-SQL, которая делит результирующий набор на группы по определенному столбцу.

Для чего используются операторы INTERSECT, EXCEPT в Transact-SQL?

Оператор EXCEPT возвращает уникальные записи из левого входного запроса, которые не выводятся правым входным запросом.

Оператор INTERSECT возвращает уникальные записи, выводимые левым и правым входными запросами.

Напишите запрос...

```
CREATE TABLE table (
 id BIGINT(20) NOT NULL AUTO_INCREMENT,
 created TIMESTAMP NOT NULL DEFAULT 0,
 PRIMARY KEY (id)
);
```

Требуется написать запрос который вернет максимальное значение id и значение created для этого id:

```
SELECT id, created FROM table where id = (SELECT MAX(id) FROM table);
```

```
CREATE TABLE track_downloads (
 download_id BIGINT(20) NOT NULL AUTO_INCREMENT,
 track_id INT NOT NULL,
 user_id BIGINT(20) NOT NULL,
 download_time TIMESTAMP NOT NULL DEFAULT 0,
 PRIMARY KEY (download_id)
);
```

Напишите SQL-запрос, возвращающий все пары (download_count, user_count), удовлетворяющие следующему условию: user_count — общее ненулевое число пользователей, сделавших ровно download_count скачиваний 19 ноября 2010 года:

```
SELECT DISTINCT download_count, COUNT(*) AS user_count
FROM (
   SELECT COUNT(*) AS download_count
   FROM track_downloads WHERE download_time=»2010-11-19»
   GROUP BY user_id)
AS download_count
GROUP BY download_count;
```

JDBC

Что такое JDBC?

JDBC, Java DataBase Connectivity (соединение с базами данных на Java) — промышленный стандарт взаимодействия Java-приложений с различными СУБД. Реализован в виде пакета java.sql, входящего в состав Java SE.

JDBC основан на концепции драйверов, которые позволяют получать соединение с базой данных по специально описанному URL. При загрузке драйвер регистрирует себя в системе и в дальнейшем автоматически вызывается, когда программа требует URL, содержащий протокол, за который этот драйвер отвечает.

В чем заключаются преимущества использования JDBC?

Преимуществами JDBC считают:

Лёгкость разработки: разработчик может не знать специфики базы данных, с которой работает;
Код практически не меняется, если компания переходит на другую базу данных (количество изменений зависит исключительно от различий между диалектами SQL);
Не нужно дополнительно устанавливать клиентскую программу;
К любой базе данных можно подсоединиться через легко описываемый URL.

Что из себя представляет JDBC URL?

JDBC URL состоит из:

<protocol>: (протокола) - всегда jdbc:.
<subprotocol>: (подпротокола) - это имя драйвера или имя механизма соединения с базой данных. Подпротокол может поддерживаться одним или несколькими драйверами. Лежащий на поверхности пример подпротокола - это «odbc», отведенный для URL, обозначающих имя источника данных ODBC. В случае необходимости использовать сервис имен (т.е. имя базы данных в JDBC URL не будет действительным именем базы данных), то подпротоколом может выступать сервис имен.
<subname> (подимени) - это идентификатор базы данных. Значение подимени может менятся в зависимости от подпротокола, и может также иметь под-подимя с синтаксисом, определяемым разработчиком драйвера. Назначение подимени - это предоставление всей информации, необходимой для поиска базы данных. Например, если база данных находится в Интернет, то в состав подимени JDBC URL должен быть включен сетевой адрес, подчиняющийся следующим соглашениям: //<hostname>:<port>/<subsubname.

Пример JDBC URL для подключения к MySQL базе данных «Test» расположенной по адресу localhost и ожидающей соединений по порту 3306: jdbc:mysql://localhost:3306/Test

Из каких частей стоит JDBC?

JDBC состоит из двух частей:

- JDBC API, который содержит набор классов и интерфейсов, определяющих доступ к базам данных. Эти классы и методы объявлены в двух пакетах - java.sql и javax.sql;
- JDBC-драйвер, компонент, специфичный для каждой базы данных.

JDBC превращает вызовы уровня API в «родные» команды того или иного сервера баз данных.

Перечислите основные классы и интерфейсы JDBC.

java.sql.DriverManager - позволяет загрузить и зарегистрировать необходимый JDBC-драйвер, а затем получить соединение с базой данных.

javax.sql.DataSource - решает те же задачи, что и DriverManager, но более удобным и универсальным образом. Существуют также javax.sql.ConnectionPoolDataSource и javax.sq1.XADataSource задача которых - обеспечение поддержки пула соединений.

java.sql.Connection - обеспечивает формирование запросов к источнику данных и управление транзакциями. Также предусмотрены интерфейсы javax.sql.PooledConnection и javax.sql.XAConnection.

java.sql.Statement , java.sql.PreparedStatement и java.sql.CallableStatement - эти интерфейсы позволяют отправить запрос к источнику данных.

java.sql.ResultSet - объявляет методы, которые позволяют перемещаться по набору данных и считывать значения отдельных полей в текущей записи.

java.sql.ResultSetMetaData - позволяет получить информацию о структуре набора данных.

java.sql.DatabaseMetaData - позволяет получить информацию о структуре источника данных.

Перечислите основные типы данных используемые в JDBC. Как они связаны с типами Java?

JDBC Type Java Object Type
CHAR String
VARCHAR String
LONGVARCHAR String
NUMERIC java.math.BigDecimal
DECIMAL java.math.BigDecimal
BIT Boolean
TINYINT Integer
SMALLINT Integer
INTEGER Integer
BIGINT Long
REAL Float
FLOAT Double
DOUBLE Double
BINARY byte[]
VARBINARY byte[]
LONGVARBINARY byte[]
DATE java.sql.Date
TIME java.sql.Time
TIMESTAMP java.sql.Timestamp
CLOB Clob
BLOB Blob
ARRAY Array
STRUCT Struct
REF Ref
DISTINCT сопоставление базового типа
JAVA_OBJECT базовый класс Java

Опишите основные этапы работы с базой данных при использовании JDBC.

1. Регистрация драйверов;
2. Установление соединения с базой данных;
3. Создание запроса(ов) к базе данных;
4. Выполнение запроса(ов) к базе данных;
5. Обработка результата(ов);
6. Закрытие соединения с базой данных.

Как зарегистрировать драйвер JDBC?

Регистрацию драйвера можно осуществить несколькими способами:

java.sql.DriverManager.registerDriver(%объект класса драйвера%).

Class.forName(«полное имя класса драйвера»).newInstance().

Class.forName(«полное имя класса драйвера»);

Как установить соединение с базой данных?

Для установки соединения с базой данных используется статический вызов java.sql.DriverManager.getConnection(...) .

В качестве параметра может передаваться:

URL базы данных

static Connection getConnection(String url)

URL базы данных и набор свойств для инициализации

static Connection getConnection(String url, Properties info)

URL базы данных, имя пользователя и пароль

static Connection getConnection(String url, String user, String password)

В результате вызова будет установлено соединение с базой данных и создан объект класса java.sql.Connection - своеобразная «сессия», внутри контекста которой и будет происходить дальнейшая работа с базой данных.

Какие уровни изоляции транзакций поддерживаются в JDBC?

Уровень изолированности транзакций — значение, определяющее уровень, при котором в транзакции допускаются несогласованные данные, то есть степень изолированности одной транзакции от другой. Более высокий уровень изолированности повышает точность данных, но при этом может снижаться количество параллельно выполняемых транзакций. С другой стороны, более низкий уровень изолированности позволяет выполнять больше параллельных транзакций, но снижает точность данных.
Во время использования транзакций, для обеспечения целостности данных, СУБД использует

блокировки, чтобы заблокировать доступ других обращений к данным, участвующим в транзакции. Такие блокировки необходимы, чтобы предотвратить:

- «грязное» чтение (dirty read) — чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится);

- неповторяющееся чтение (non-repeatable read) — при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными;

- фантомное чтение (phantom reads) — ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк.

Уровни изоляции транзакций определены в виде констант интерфейса java.sql.Connection:

- TRANSACTION_NONE – драйвер не поддерживает транзакции;

- TRANSACTION_READ_UNCOMMITTED – позволяет транзакциям видеть несохраненные изменения данных: разрешает грязное, непроверяющееся и фантомное чтения;

- TRANSACTION_READ_COMMITTED – любое изменение, сделанное в транзакции, не видно вне неё, пока она не сохранена: предотвращает грязное чтение, но разрешает непроверяющееся и фантомное;

- TRANSACTION_REPEATABLE_READ – запрещает грязное и непроверяющееся, фантомное чтение разрешено;

- TRANSACTION_SERIALIZABLE – грязное, непроверяющееся и фантомное чтения запрещены.

NB! Сервер базы данных может не поддерживать все уровни изоляции. Интерфейс java.sql.DatabaseMetaData предоставляет информацию об уровнях изолированности транзакций, которые поддерживаются данной СУБД.

Уровень изоляции транзакции используемый СУБД можно задать с помощью метода setTransactionIsolation() объекта java.sql.Connection. Получить информацию о применяемом уровне изоляции поможет метод getTransactionIsolation().

При помощи чего формируются запросы к базе данных?

Для выполнения запросов к базе данных в Java используются три интерфейса:

java.sql.Statement - для операторов SQL без параметров;
java.sql.PreparedStatement - для операторов SQL с параметрами и часто выполняемых операторов;
java.sql.CallableStatement - для исполнения хранимых в базе процедур.

Объекты-носители интерфейсов создаются при помощи методов объекта java.sql.Connection:

java.sql.createStatement() возвращает объект Statement;
java.sql.prepareStatement() возвращает объект PreparedStatement;
java.sql.prepareCall() возвращает объект CallableStatement;

Чем отличается Statement от PreparedStatement?

Statement: используется для простых случаев запроса без параметров.
PreparedStatement: предварительно компилирует запрос, который может содержать входные пара-

метры и выполняться несколько раз с разным набором этих параметров.

Перед выполнением СУБД разбирает каждый запрос, оптимизирует его и создает «план» (query plan) его выполнения. Если один и тот же запрос выполняется несколько раз, то СУБД в состоянии кэшировать план его выполнения и не производить этапов разборки и оптимизации повторно. Благодаря этому запрос выполняется быстрее.

Суммируя: PreparedStatement выгодно отличается от Statement тем, что при повторном использовании с одним или несколькими наборами параметров позволяет получить преимущества заранее прекомпилированного и кэшированного запроса, помогая при этом избежать SQL Injection.

Как осуществляется запрос к базе данных и обработка результатов?

Выполнение запросов осуществляется при помощи вызова методов объекта, реализующего интерфейс java.sql.Statement:

executeQuery() - для запросов, результатом которых является один набор значений, например запросов SELECT. Результатом выполнения является объект класса java.sql.ResultSet;

executeUpdate() - для выполнения операторов INSERT, UPDATE или DELETE, а также для операторов DDL (Data Definition Language). Метод возвращает целое число, показывающее, сколько записей было модифицировано;

execute() – исполняет SQL-команды, которые могут возвращать различные результаты. Например, может использоваться для операции CREATE TABLE. Возвращает true, если первый результат содержит ResultSet и false, если первый результат - это количество модифицированных записей или результат отсутствует. Чтобы получить первый результат необходимо вызвать метод getResultSet() или getUpdateCount(). Остальные результаты доступны через вызов getMoreResults(), который при необходимости может быть произведён многократно.

Объект с интерфейсом java.sql.ResultSet хранит в себе результат запроса к базе данных - некий набор данных, внутри которого есть курсор, указывающий на один из элементов набора данных - текущую запись.

Используя курсор можно перемещаться по набору данных при помощи метода next().

NB! Сразу после получения набора данных его курсор находится перед первой записью и чтобы сделать её текущей необходимо вызвать метод next().

Содержание полей текущей записи доступно через вызовы методов getInt(), getFloat(), getString(), getDate() и им подобных.

Как вызвать хранимую процедуру?

Хранимые процедуры – это именованный набор операторов SQL хранящийся на сервере. Такую процедуру можно вызвать из Java-класса с помощью вызова методов объекта реализующего интерфейс java.sql.Statement.

Выбор объекта зависит от характеристик хранимой процедуры:

без параметров → Statement
с входными параметрами → PreparedStatement
с входными и выходными параметрами → CallableStatement

Если неизвестно, как была определена хранимая процедура, для получения информации о хранимой процедуре (например, имен и типов параметров) можно использовать методы java.sql. DatabaseMetaData позволяющие получить информацию о структуре источника данных.

Пример вызова хранимой процедуры с входными и выходными параметрами:

```
public vois runStoredProcedure(final Connection connection) throws Exception {
  // описываем хранимую процедуру
  String procedure = «{ call procedureExample(?, ?, ?) }»;

  // подготавливаем запрос
  CallableStatement cs = connection.prepareCall(procedure);

  // устанавливаем входные параметры
  cs.setString(1, «abcd»);
  cs.setBoolean(2, true);
  cs.setInt(3, 10);

  // описываем выходные параметры
  cs.registerOutParameter(1, java.sql.Types.VARCHAR);
  cs.registerOutParameter(2, java.sql.Types.INTEGER);

  // запускаем выполнение хранимой процедуры
  cs.execute();

  // получаем результаты
  String parameter1 = cs.getString(1);
  int parameter2 = cs.getInt(2);

  // заканчиваем работу с запросом
  cs.close();
}
```

Как закрыть соединение с базой данных?

Соединение с базой данной закрывается вызовом метода close() у соответствующего объекта java. sql.Connection или посредством использования механизма try-with-resources при создании такого объекта, появившегося в Java 7.

NB! Предварительно необходимо закрыть все запросы созданные этим соединением.

Тестирование

Что такое «модульное тестирование»?

Модульное/компонентное тестирование (unit testing) - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Модульные тесты можно условно поделить на две группы:

1. тесты состояния (state based), проверяющие что вызываемый метод объекта отработал корректно, проверяя состояние тестируемого объекта после вызова метода.

2. тесты взаимодействия (interaction tests), в которых тестируемый объект производит манипуляции с другими объектами. Применяются, когда требуется удостовериться, что тестируемый объект корректно взаимодействует с другими объектами.

Что такое «интеграционное тестирование»?

Интеграционное тестирование (integration testing) — это тестирование, проверяющие работоспособность двух или более модулей системы в совокупности — то есть нескольких объектов как единого блока. В тестах взаимодействия же тестируется конкретный, определенный объект и то, как именно он взаимодействует с внешними зависимостями.

Чем интеграционное тестирование отличается от модульного?

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа.

Допустим, есть класс, который при определенных условиях взаимодействует с web-сервисом через зависимый объект. И нам надо проверить, что определенный метод зависимого объекта действительно вызывается. Если в качестве зависимого класса передать:

- реальный класс, работающий с web-сервисом, то это будет интеграционное тестирование.

- заглушку, то это будет тестирование состояния.

- шпиона, а в конце теста проверить, что определенный метод зависимого объекта действительно был вызван, то это будет тест взаимодействия.

Какие существуют виды тестовых объектов?

- пустышка (dummy) - объект, который обычно передается в тестируемый класс в качестве параметра, но не имеет поведения: с ним ничего не происходит и никакие его методы не вызываются.

Примером dummy-объектов являются new object(), null, «Ignored String» и т.д.
- фальшивка (fake object) применяется в основном для ускорения запуска ресурсоёмких тестов и

является заменой тяжеловесного внешнего зависимого объекта его легковесной реализацией.

Основные примеры — эмулятор базы данных (fake database) или фальшивый web-сервис.

- заглушка (test stub) используется для получения данных из внешней зависимости, подменяя её. При этом заглушка игнорирует все данные поступающие из тестируемого объекта, возвращая заранее определённый результат.

Тестируемый объект использует чтение из конфигурационного файла? Тогда передаем ему заглушку ConfigFileStub возвращающую тестовые строки конфигурации без обращения к файловой системе.

- шпион (test spy) - разновидность заглушки, которая умеет протоколировать сделанные к ней обращения из тестируемой системы, чтобы проверить их правильность в конце теста. При этом фиксируется количество, состав и содержание параметров вызовов.

Если существует необходимость проверки, что определённый метод тестируемого класса вызывался ровно 1 раз, то шпион - имменно то, что нам нужно.

- фикция (mock object) похож на шпиона, но обладает расширенной функциональностью, заранее заданными поведением и реакцией на вызовы.

Чем stub отличается от mock?

stub используется как заглушка сервисов, методов, классов и т.д. с заранее запрограммированным ответом на вызовы.

mock использует подмену результатов вызова, проверяет сам факт взаимодействия, протоколирует и контролирует его.

Что такое «фикстуры»?

Фикстуры (fixtures) - состояние среды тестирования, которое требуется для успешного выполнения теста. Основная задача фикстур заключается в подготовке тестового окружения с заранее фиксированным/известным состоянием, чтобы гарантировать повторяемость процесса тестирования.

Какие аннотации фикстур существуют в JUnit?

@BeforeClass - определяет код, который должен единожды выполниться перед запуском набора тестовых методов.
@AfterClass - код, выполняемый один раз после исполнения набора тестовых методов.
@Before - определяет код, который должен выполняться каждый раз перд запуском любого тестовым методом.
@After - код, выполняемый каждый раз после исполнения любого тестового метода.

Для чего в JUnit используется аннотация @Ignore?

@Ignore указывает JUnit на необходимость пропустить данный тестовый метод.

Журналирование

Какие существуют типы логов?

- системы (System);
- безопасности (Security);
- приложения (Application, Buisness).

Пользователь входит в приложение, проверяется пароль. Это действие относится к безопасности (Security). Дальше он запускает какой-нибудь модуль. Это событие уровня приложения (Application). Модуль при старте обращается к другому модулю за какими-то дополнительными данными, производит какие-либо еще вызовы – это уже системные действия (System).

Из каких частей состоит система журналирования log4j?

Система журналирования состоит из трёх основных частей:

- управляющей журналированием - logger;
- добавляющей в журнал - appender;
- определяющей формат добавления - layout.

Что такое Logger в log4j?

Logger представляет собой объект класса org.apache.log4j.Logger, который используется как управляющий интерфейс для журналирования сообщений с возможностью задавать уровень детализации. Именно logger проверяет нужно ли обрабатывать сообщение и если журналирование необходимо, то сообщение передаётся в appender, если нет - система завершает обработку данного сообщения.

Что такое Appender в log4j?

Appender - это именованный объект журнала событий, реализующий интерфейс org.apache.log4j. Appender и добавляющий события в журнал. Appender вызывает разные вспомогательные инструменты - компоновщик, фильтр, обработчик ошибок (если они определены и необходимы). В ходе этой работы окончательно устанавливается необходимость записи сообщения, сообщению придаются окончательные содержание и форма.

В log4j журнал может представлять:

консоль;
файл;
сокет;
объект класса реализующего java.io.Writer или java.io.OutputStream;
JDBC хранилище;
тему (topic) JMS;
NT Event Log;
SMTP;
Syslog;
Telnet.

Наиболее часто используемые log4j appender-ы:

org.apache.log4j.ConsoleAppender - вывод в консоль;
org.apache.log4j.FileAppender - добавление в файл;
org.apache.log4j.DailyRollingFileAppender - добавление в файл с обновлением файла через задан-

ный промежуток времени;
org.apache.log4j.RollingFileAppender - добавление в файл с обновлением файла по достижению определенного размера;
org.apache.log4j.varia.ExternallyRolledFileAppender - расширение RollingFileAppender обновляющее файл по команде принятой с заданного порта;
org.apache.log4j.net.SMTPAppender - сообщение по SMTP;
org.apache.log4j.AsyncAppender - позволяет, используя отдельный поток, организовать асинхронную работу, когда сообщения фиксируются лишь при достижении определенного уровня заполненности промежуточного буфера.
org.apache.log4j.nt.NTEventLogAppender - добавление в NT Event Log;
org.apache.log4j.net.SyslogAppender - добавление в Syslog;
org.apache.log4j.jdbc.JDBCAppender - запись в хранилище JDBC;
org.apache.log4j.lf5.LF5Appender - сообщение передаётся в специальный GUI интерфейс LogFactor5
org.apache.log4j.net.SocketAppender - трансляция сообщения по указанному адресу и порту;
org.apache.log4j.net.SocketHubAppender - рассылка сообщения сразу нескольким удалённым серверам соединённым по заданному порту;
org.apache.log4j.net.TelnetAppender - отсылка сообщения по протоколу Telenet;
org.apache.log4j.net.JMSAppender - добавление сообщения в JMS.

Что такое Layout в log4j?

Layout - наследник класса org.apache.log4j.Layout предоставляющий возможность форматирования сообщения перед добавлением в журнал.

В log4j существуют следующие типы layout-ов:

org.apache.log4j.SimpleLayout - на выходе получается строка содержащая лишь уровень вывода и сообщение;
org.apache.log4j.HTMLLayout - форматирует сообщение в виде элемента HTML-таблицы;
org.apache.log4j.xml.XMLLayout - компанует сообщение в виде XML формате;
org.apache.log4j.TTCCLayout - на выходе сообщение дополняется информацией о времени, потоке, имени логгера и вложенном диагностическом контексте;
org.apache.log4j.PatternLayout / org.apache.log4j.EnhancedPatternLayout - настройка форматирования сообщения при помощи шаблона заданного пользователем.

Перечислите уровни журналирования в log4j? Назовите порядок их приоритетности.

OFF - отсутствие журналирования;
FATAL - фатальная ошибка;
ERROR - ошибка;
WARN - предупреждение;
INFO - информация;
DEBUG - детальная информация для отладки;
TRACE – трассировка всех сообщений.

Между уровнями логирования установлен следующий порядок приоритетов:

OFF < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < ALL

Какие существуют способы конфигурирования log4j?

Для того, чтобы log4j начал работать нужно предоставить ему конфигурацию. Это можно сделать несколькими путями:

- Создать конфигурацию программно, т.е. получить logger, определить уровень журналирования, прикрепить appender и задать способ форматирования.
- Указать файл или URL как аргумент при запуске java-машины -Dlog4j.configuration=путь/к/файлу/конфигурации, а затем прочитать его в программе при помощи PropertyConfigurator.configure(...)/DOMConfigurator.configure(...) для формата .properties или XML соответственно.
- Загрузить конфигурацию из файла в формате XML или .properties: log4j ищет файл конфигурации в classpath. Сначала ищется файл log4j.xml и, если таковой не найден, - файл log4j.properties.

UML

Что такое UML?

UML – это унифицированный графический язык моделирования для описания, визуализации, проектирования и документирования объектно-ориентированных систем. UML призван поддерживать процесс моделирования на основе объектно-ориентированного подхода, организовывать взаимосвязь концептуальных и программных понятий, отражать проблемы масштабирования сложных систем.

Отличительной особенностью UML является то, что словарь этого языка образуют графические элементы. Каждому графическому символу соответствует конкретная семантика, поэтому модель, созданная одним человеком, может однозначно быть понята другим человеком или программным средством, интерпретирующим UML. Отсюда, в частности, следует, что модель системы, представленная на UML, может автоматически быть переведена на объектно-ориентированный язык программирования, то есть, при наличии хорошего инструментального средства визуального моделирования, поддерживающего UML, построив модель, мы получим и заготовку программного кода, соответствующего этой модели.

Что такое «диаграмма», «нотация» и «метамодель» в UML?

Диаграмма - графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика

Нотация – совокупность символов и правила их применения, используются для представления понятий и связей между ними. Нотация диаграммы определяет способ представления, ассоциации, множественности. Причем эти понятия должны быть точно определены.

Метамодель – диаграмма, определяющая нотацию. Метамодель помогает понять, что такое хорошо организованная, т.е. синтаксически правильная, модель.

Какие существуют типы диаграмм?

Структурные диаграммы:

1. классов (Class diagram) описывает структуру системы, демонстрирующая классы системы, их атрибуты, методы и зависимости между классами.

2. объектов (Object diagram) демонстрирует полный или частичный снимок моделируемой системы в заданный момент времени. На диаграмме объектов отображаются экземпляры классов (объекты) системы с указанием текущих значений их атрибутов и связей между объектами.

3. компонентов (Component diagram) показывает разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами.

- развёртывания/размещения (Deployment diagram) служит для моделирования работающих узлов и артефактов, развёрнутых на них.

- пакетов (Package diagram) используется для организации элементов в группы по какому-либо признаку с целью упрощения структуры и организации работы с моделью системы.

- профилей (Profile diagram) действует на уровне метамодели и показывает стереотип класса или пакета.
- композитной/составной структуры (Composite structure diagram) демонстрирует внутреннюю

структуру класса и, по возможности, взаимодействие элементов (частей) его внутренней структуры.

- кооперации (Collaboration diagram) показывает роли и взаимодействие классов в рамках кооперации.

Диаграммы поведения:

1. деятельности (Activity diagram) показывает разложение некоторой деятельности на её составные части. Под деятельностью понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий, соединённых между собой потоками, которые идут от выходов одного узла к входам другого. Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.

2. состояний/автомата/конечного автомата (State Machine diagram) представляет конечный автомат с простыми состояниями, переходами и композитными состояниями. Конечный автомат (State machine) — спецификация последовательности состояний, через которые проходит объект или взаимодействие в ответ на события своей жизни, а также ответные действия объекта на эти события. Конечный автомат прикреплён к исходному элементу (классу, кооперации или методу) и служит для определения поведения его экземпляров.

3. вариантов использования/прецедентов (Use case diagram) отражает отношения существующие между актёрами и вариантами использования. Основная задача — представлять собой единое средство, дающее возможность заказчику, конечному пользователю и разработчику совместно обсуждать функциональность и поведение системы.

4. взаимодействия (Interaction diagram):

- коммуникации (Communication diagram) изображает взаимодействия между частями композитной структуры или ролями кооперации при этом явно указываются отношения между элементами (объектами), а время как отдельное измерение не используется (применяются порядковые номера вызовов).

- последовательности (Sequence diagram) показывает взаимодействия объектов, упорядоченные по времени их проявления.

- обзора взаимодействия (Interaction overview diagram) — разновидность диаграммы деятельности, включающая фрагменты диаграммы последовательности и конструкции потока управления.

- синхронизации (Timing diagram) — альтернативное представление диаграммы последовательности, явным образом показывающее изменения состояния на линии жизни с заданной шкалой времени. Может быть полезна в приложениях реального времени.

Какие виды отношений существуют в структурной диаграмме классов?

Взаимосвязи классов

1. Обобщение (Generalization) показывает, что один из двух связанных классов (подтип) является частной формой другого (супертипа), который называется обобщением первого. На практике это означает, что любой экземпляр подтипа является также экземпляром супертипа. Обобщение также известно как наследование, «is a» взаимосвязь или отношение «является».

«Табурет» является подтипом «Мебели».

2. Реализация (Implementation) — отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). Реализация — отношение целое-часть. Поставщик, как правило является абстрактным классом или классом-интерфейсом.

«Кровать» реализует поведение «Мебели для сна»

Взаимосвязи объектов классов:

1. Зависимость (Dependency) обозначает такое отношение между классами, что изменение спецификации класса-поставщика может повлиять на работу зависимого класса, но не наоборот.

«Расписание занятий» имеет зависимость от «Списка предметов». При изменении списка предметов расписание занятий будет вынуждено изменится. Однако изменение расписания занятий никак не влияет на список предметов.

2. Ассоциация (Association) показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому. Является общим случаем композиции и агрегации.

«Студент» и «Университет» имеют ассоциацию т.к. студент может учиться в университете и этой ассоциации можно присвоить имя «учится в».

3. Агрегация (Aggregation) — это разновидность ассоциации в отношении между целым и его частями. Как тип ассоциации агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое). Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

«Студент» не является неотъемлемой частью «Группы», но в то же время, группа состоит из студентов, поэтому следует использовать агрегацию.

4. Композиция (Composition) — более строгий вариант агрегации. Известна также как агрегация по значению. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

«Факультет» является частью «Университета» и факультет без университета существовать не может, следовательно здесь подходит композиция.

#Общие взаимосвязи

1. Зависимость — это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причём обратное не обязательно. Возникает, когда объект выступает, например, в форме параметра или локальной переменной. Существует несколько именованных вариантов. Зависимость может быть между экземплярами, классами или экземпляром и классом.

2. Уточнение отношений имеет отношение к уровню детализации. Один пакет уточняет другой, если в нём содержатся те же самые элементы, но в более подробном представлении.

3. Мощность/кратность/мультипликатор отношения означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце. Различают следующие типичные случаи:

нотация	объяснение	пример
0..1	Ноль или один экземпляр	Кошка имеет или не имеет хозяина
1	Обязательно один экзмепляр	У кошки одна мать
0..* или *	Ноль или более экзмепляров	У кошки могут быть, а может и не быть котят
1..*	Один или более экземпляров	У кошки есть хотя бы одно место, где она спит

XML

Что такое XML?

XML, eXtensible Markup Language (расширяемый язык разметки) - язык с простым формальным синтаксисом, хорошо приспособленный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком.

XML расширяем, он не фиксирует разметку, используемую в документах и разработчик волен создавать разметку в соответствии с потребностями конкретной области, будучи ограниченным лишь синтаксическими правилами языка.

Что такое DTD?

DTD, Document Type Definition (определение типа документа) — это заранее определённый свод правил, задающий связи между элементами и атрибутами.

Например, DTD для HTML гласит, что тэг DIV должен быть внутри тэга BODY и может встречаться многократно, TITLE — в HEAD и всего один раз, а SCRIPT – и там, и там сколь угодно раз.

DTD обычно описывается непосредственно в документе в виде строки-формулировки, начинающейся с <!DOCTYPE ... > или отдельном файле.

Чем well-formed XML отличается от valid XML?

В зависимости от уровня соответствия стандартам документ может быть «well-formed» («правильно построенный»), либо «valid» («действительный»).

Основные признаки well-formed XML следуют из формального описания стандарта:

- Документ имеет ровно один корневой элемент, в котором лежат все остальные. То есть, <document>...</document><appendix>...</appendix> - это не XML-документ.
- Все открытые теги обязаны быть закрыты. HTML, например, допускает не закрывать многие теги (<p>, <body>, <li>, <td> и многие другие). В XML так делать нельзя.
- Для одиночных тегов (типа
) , чтобы отличать их от открывающих, предусмотрена специальная запись:
. Но можно написать и полностью
</br>.
- Имена тегов регистрозависимые. Если вы открываете тег <SiteDescription>, то его надо закрывать именно таким же, </sitedescription> не допускается.
- Теги не могут нарушать вложенность. Вот такого не должно быть: ....
- Все атрибуты тегов обязаны быть заключены в двойные кавычки («).
- Есть три символа - <, > и &, которые обязаны быть экранированы везде с помощью <, > и &. Внутри атрибутов надо экранировать еще и двойную кавычку с помощью ".
-Все символы в документе обязаны соответствовать заявленной кодировке.

Документ является valid, если он сформирован с соблюдением всех синтаксических правил корректности конкретного XML, т.е. соответствует DTD.

well-formed XML - корректен синтаксически (может быть разобран парсером), а valid XML - корректен как синтаксически так и семантически (удовлетворяет правилам заранее описанных словаря и грамматики (DTD)).

Что такое «пространство имен» в XML?

Пространство имён XML (XML namespace) - это идентифицируемая с помощью ссылки URI коллекция имен, используемых в XML документах для обозначения типов элементов и именования атри-

бутов. Пространство имен XML отличается от тех «пространств имен», которые обычно используются в компьютерных дисциплинах, тем, что в варианте для XML оно имеет внутреннюю структуру, и, с математической точки зрения, набором не является.

Пространства имён объявляются с помощью XML атрибута xmlns, значением которого должен быть URI и префикса, однозначно идентифицирующего пространство имён каждого элемента.

Все имена элементов в пределах пространства имён должны быть уникальны.

В общем случае пространство имён XML не требует, чтобы был определён его словарь.

XML-документ может содержать имена элементов и атрибутов из нескольких словарей XML. В каждом словаре задано своё пространство имён — так разрешается проблема неоднозначности имён элементов и атрибутов.

Что такое XSD? В чём его преимущества перед XML DTD?

XSD, XML Schema Definition, XML Schema (XML схема) — язык описания структуры XML-документа. В частности, XML Schema описывает:

- словарь - имена элементов и атрибутов;
- модель содержания - взаимосвязи между элементами и атрибутами, а также их
- структуру документа;
- используемые типы данных.

Преимущества XSD перед DTD заключаются в следующем:

- DTD, в отличии от XSD, не является XML и имеет свой собственный синтаксис. В связи с этим могут возникать разнообразные проблемы с кодировкой и верификацией XML-документов.

- При использовании XSD XML-парсер может проверить не только правильность синтаксиса XML документа, но также его структуру, модель содержания и типы данных. В XML DTD существует лишь один тип данных – строка и если, например, в числовом поле будет текст, то документ всё же сможет пройти верификацию, так как XML DTD не сможет проверить тип данных.

- Нельзя поставить в соответствие одному XML документу больше одного DTD. А следовательно и верифицировать документ можно лишь одним DTD описанием. XSD расширяем, и позволяет подключать несколько словарей для описания типовых задач.

- XSD обладает встроенными средствами документирования, позволяющими создавать самодостаточные документы, не требующие дополнительного описания.

Какие типы существуют в XSD?

Простой тип - это определение типа для значения, которое может использоваться в качестве содержимого элемента или атрибута. Этот тип данных не может содержать элементы или иметь атрибуты.

```
<xsd:element name='price' type='xsd:decimal'/>
...
<price>45.50</price>
```

Сложный тип - это определение типа для элементов, которые могут содержать атрибуты и другие

элементы.

```
<xsd:element name='price'>
  <xsd:complexType base='xsd:decimal'>
    <xsd:attribute name='currency' type='xsd:string'/>
  </xsd:complexType>
</xsd:element>
...
<price currency='US'>45.50</price>
```

Какие вы знаете методы чтения XML? Опишите сильные и слабые стороны каждого метода.

DOM (Document Object Model) - объектный - считывает XML, воссоздавая его в памяти в виде объектной структуры при этом XML документ представляется в виде набора тегов – узлов. Каждый узел может иметь неограниченное количество дочерних узлов. Каждый дочерний тоже может содержать несколько уровней потомков или не содержать их вовсе. Таким образом в итоге получается некое дерево.

- Низкая скорость работы.

- Расходует много памяти.

+ Прост в программировании.

+ Если в XML много объектов с перекрёстными ссылками друг на друга, достаточно дважды пройтись по документу: первый раз создать объекты без ссылок и заполнить словарь «название-объект», второй раз — восстановить ссылки.

+ При ошибке в XML в памяти остаётся полусозданная структура XML, которая будет автоматически уничтожена.

+ Пригоден как для чтения так и для записи.

SAX (Simple API for XML) событийный - читает XML документ, реагируя на появляющиеся события (открывающий или закрывающий тег, строку, атрибут) вызовом предоставляемых приложением обработчиков событий. При этом, в отличии от DOM, не сохраняет документ в памяти.

+ Высокая скорость работы

+ Расходует мало памяти.

\ Довольно сложен в программировании.

- Если в XML много объектов с перекрёстными ссылками друг на друга, надо организовать временное хранение строковых ссылок, чтобы потом, когда документ будет считан, преобразовать в указатели.

- При ошибке в XML в памяти остаётся полусозданная структура предметной отрасли; программист должен своими руками корректно уничтожить её.

- Пригоден только для чтения.

StAX (Stream API for XML) потоковый - состоящий из двух наборов API для обработки XML, ко-

торые обеспечивают разные уровни абстракции. API с использованием курсора позволяет приложениям работать с XML как с потоком лексем (или событий); приложение может проверить статус анализатора и получить информацию о последней проанализированной лексеме, а затем перейти к следующей. Второй, высокоуровневый API, использующий итераторы событий, позволяет приложению обрабатывать XML как серию объектов событий, каждый из которых взаимодействует с фрагментом XML-структуры приложения. Всё, что требуется от приложения - это определить тип синтаксически разобранного события, отнести его к соответствующему конкретному типу и использовать соответствующие методы для получения информации, относящейся к событию.

\ Сохраняет преимущества, которые есть в SAX по сравнению с DOM.

\+ Не основан на обратных вызовах обработчиков, приложению не придется обслуживать эмулированное состояние анализатора, как это происходит при использовании SAX.

\- Пригоден только для чтения.

Когда следует использовать DOM, а когда SAX, StAX анализаторы?

DOM - естественный выбор, когда объектом предметной области является сам XML: когда нужно знать и иметь возможность изменять структуру документа, а также в случае многократного использования информации из документа.

Для быстрого одноразового чтения оптимальным является использование SAX или StAX.

Какие вы знаете способы записи XML?

Прямая запись - пишет XML тег за тегом, атрибут за атрибутом.

\+ Высокая скорость работы.

\+ Экономия памяти: при использовании не создаётся промежуточных объектов.

\- Пригоден только для записи.

Запись DOM (Document Object Model) - создаёт полную структуру XML и только потом записывает её.

\- Низкая скорость работы.

\- Не оптимальный расход памяти.

\+ Пригоден как для записи так и для чтения.

Что такое JAXP?

JAXP, The Java API for XML Processing (Java API для обработки XML) — набор API, упрощающих обработку XML данных в программах написанных на Java. Содержит реализации DOM, SAX и StAX парсеров, поддерживает XSLT и возможность работать с DTD.

Что такое XSLT?

XSLT, eXtensible Stylesheet Language Transformations — язык преобразования XML-документов.

XSLT создавался для применения в XSL (eXtensible Stylesheet Language) - языке стилей для XML. Во время XSL-преобразования XSLT-процессор считывает XML-документ и таблицу(ы) стилей XSLT. На основе инструкций, которые процессор находит в таблице(ах) стилей XSLT, он вырабатывает новый XML-документ или его фрагмент.

Шаблоны проектирования

Что такое «шаблон проектирования»?

Шаблон (паттерн) проектирования (design pattern) — это проверенное и готовое к использованию решение. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее - он не зависит от языка программирования, не является законченным образцом, который может быть прямо преобразован в код и может быть реализован по разному в разных языках программирования.

Плюсы использования шаблонов:

- снижение сложности разработки за счёт готовых абстракций для решения целого класса проблем.
- облегчение коммуникации между разработчиками, позволяя ссылаться на известные шаблоны.
- унификация деталей решений: модулей и элементов проекта.
- возможность отыскав удачное решение, пользоваться им снова и снова.
- помощь в выборе выбрать наиболее подходящего варианта проектирования.

Минусы:

- слепое следование некоторому выбранному 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 назначение данного шаблона;
- Задача - задача, которую шаблон позволяет решить;
- Способ решения - способ, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден;
- Участники - сущности, принимающие участие в решении задачи;
- Следствия- последствия от использования шаблона как результат действий, выполняемых в шаблоне;
- Реализация - возможный вариант реализации шаблона.

Типы шаблонов проектирования.

- Основные (Fundamental) - основные строительные блоки других шаблонов. Большинство других шаблонов использует эти шаблоны в той или иной форме.
- Порождающие шаблоны (Creational) — шаблоны проектирования, которые абстрагируют процесс создание экземпляра. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять созданный объект, а шаблон, порождающий объекты, делегирует создание объектов другому объекту.
- Структурные шаблоны (Structural) определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.
- Поведенческие шаблоны (Behavioral) определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Приведите примеры основных шаблонов проектирования.

- Делегирование (Delegation pattern) - Сущность внешне выражает некоторое поведение, но в ре-

альности передаёт ответственность за выполнение этого поведения связанному объекту.

- Функциональный дизайн (Functional design) - Гарантирует, что каждая сущность имеет только одну обязанность и исполняет её с минимумом побочных эффектов на другие.
- Неизменяемый интерфейс (Immutable interface) - Создание неизменяемого объекта.
- Интерфейс (Interface) - Общий метод структурирования сущностей облегчающий их понимание.
- Интерфейс-маркер (Marker interface) - В качестве атрибута (как пометки объектной сущности) применяется наличие или отсутствие реализации интерфейса-маркера. В современных языках программирования вместо этого применяются атрибуты или аннотации.
- Контейнер свойств (Property container) - Позволяет добавлять дополнительные свойства сущности в контейнер внутри себя, вместо расширения новыми свойствами.
- Канал событий (Event channel) - Создаёт централизованный канал для событий. Использует сущность-представитель для подписки и сущность-представитель для публикации события в канале. Представитель существует отдельно от реального издателя или подписчика. Подписчик может получать опубликованные события от более чем одной сущности, даже если он зарегистрирован только на одном канале.

Приведите примеры порождающих шаблонов проектирования.

- Абстрактная фабрика (Abstract factory) - Класс, который представляет собой интерфейс для создания других классов.
- Строитель (Builder) - Класс, который представляет собой интерфейс для создания сложного объекта.
- Фабричный метод (Factory method) - Делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.
- Прототип (Prototype) - Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.
- Одиночка (Singleton) - Класс, который может иметь только один экземпляр.

Приведите примеры структурных шаблонов проектирования.

- Адаптер (Adapter) - Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.
- Мост (Bridge) - Структура, позволяющая изменять интерфейс обращения и интерфейс реализации класса независимо.
- Компоновщик (Composite) - Объект, который объединяет в себе объекты, подобные ему самому.
- Декоратор (Decorator) - Класс, расширяющий функциональность другого класса без использования наследования.
- Фасад (Facade) - Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.
- Приспособленец (Flyweight) - Это объект, представляющий себя как уникальный экземпляр в разных местах программы, но по факту не являющийся таковым.
- Заместитель (Proxy) - Объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него.

Приведите примеры поведенческих шаблонов проектирования.

- Цепочка обязанностей (Chain of responsibility) - Предназначен для организации в системе уров-

ней ответственности.

- Команда (Command) - Представляет действие. Объект команды заключает в себе само действие и его параметры.
- Интерпретатор (Interpreter) - Решает часто встречающуюся, но подверженную изменениям, задачу.
- Итератор (Iterator) - Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого + __из объектов, входящих в состав агрегации.
- Посредник (Mediator) - Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.
- Хранитель (Memento) - Позволяет не нарушая инкапсуляцию зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.
- Наблюдатель (Observer) - Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.
- Состояние (State) - Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.
- Стратегия (Strategy) - Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.
- Шаблонный метод (Template method) - Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
- Посетитель (Visitor) - Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы.

Что такое «антипаттерн»? Какие антипаттерны вы знаете?

Антипаттерн (anti-pattern) — это распространённый подход к решению класса часто встречающихся проблем, являющийся неэффективным, рискованным или непродуктивным.

Poltergeists (полтергейсты) - это классы с ограниченной ответственностью и ролью в системе, чьё единственное предназначение — передавать информацию в другие классы. Их эффективный жизненный цикл непродолжителен. Полтергейсты нарушают стройность архитектуры программного обеспечения, создавая избыточные (лишние) абстракции, они чрезмерно запутанны, сложны для понимания и трудны в сопровождении. Обычно такие классы задумываются как классы-контроллеры, которые существуют только для вызова методов других классов, зачастую в предопределенной последовательности.

Признаки появления и последствия антипаттерна

- Избыточные межклассовые связи.
- Временные ассоциации.
- Классы без состояния (содержащие только методы и константы).
- Временные объекты и классы (с непродолжительным временем жизни).
- Классы с единственным методом, который предназначен только для создания или вызова других классов посредством временной ассоциации.
- Классы с именами методов в стиле «управления», такие как startProcess.

Типичные причины

- Отсутствие объектно-ориентированной архитектуры (архитектор не понимает объектно-ориентированной парадигмы).
- Неправильный выбор пути решения задачи.
- Предположения об архитектуре приложения на этапе анализа требований (до объектно-ориен-

тированного анализа) могут также вести к проблемам на подобии этого антипаттерна.

Внесенная сложность (Introduced complexity): Необязательная сложность дизайна. Вместо одного простого класса выстраивается целая иерархия интерфейсов и классов. Типичный пример «Интерфейс - Абстрактный класс - Единственный класс реализующий интерфейс на основе абстрактного».

Инверсия абстракции (Abstraction inversion): Сокрытие части функциональности от внешнего использования, в надежде на то, что никто не будет его использовать.

Неопределённая точка зрения (Ambiguous viewpoint): Представление модели без спецификации её точки рассмотрения.

Большой комок грязи (Big ball of mud): Система с нераспознаваемой структурой.

Божественный объект (God object): Концентрация слишком большого количества функций в одной части системы (классе).

Затычка на ввод данных (Input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода.

Раздувание интерфейса (Interface bloat): Разработка интерфейса очень мощным и очень сложным для реализации.

Волшебная кнопка (Magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, написание прикладной логики в обработчиках нажатий на кнопку.

Перестыковка (Re-Coupling): Процесс внедрения ненужной зависимости.

Дымоход (Stovepipe System): Редко поддерживаемая сборка плохо связанных компонентов.

Состояние гонки (Race hazard): непредвидение возможности наступления событий в порядке, отличном от ожидаемого.

Членовредительство (Mutilation): Излишнее «затачивание» объекта под определенную очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.

Сохранение или смерть (Save or die): Сохранение изменений лишь при завершении приложения.

Что такое Dependency Injection?

Dependency Injection (внедрение зависимости) - это набор паттернов и принципов разработки програмного обеспечения, которые позволяют писать слабосвязный код. В полном соответствии с принципом единой обязанности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Основы HTML

Что такое «HTML»?

HTML, HyperText Markup Language («язык гипертекстовой разметки») — стандартизированный язык разметки документов в WWW. На данный момент актуальна 5 версия этого языка - HTML5.

Что такое «XHTML»?

XHTML, eXtensible HyperText Markup Language («расширяемый язык гипертекстовой разметки») - более строгий вариант HTML, следующий всем ограничениям XML и, фактически являющийся приложением языка XML к области разметки гипертекста.

Что такое DOCTYPE и зачем он нужен?

Элемент <!DOCTYPE> предназначен для указания типа текущего документа. Это необходимо, чтобы браузер понимал согласно какого стандарта необходимо интерпретировать данную web-страницу.

Существует несколько видов <!DOCTYPE>, различающихся версией языка, на который они ориентированы:

1. HTML 4.01

<!DOCTYPE HTML PUBLIC «-//W3C//DTD HTML 4.01//EN» «http://www.w3.org/TR/html4/strict.dtd»>: строгий синтаксис HTML;

<!DOCTYPE HTML PUBLIC «-//W3C//DTD HTML 4.01 Transitional//EN» «http://www.w3.org/TR/html4/loose.dtd»>: переходный синтаксис HTML;

<!DOCTYPE HTML PUBLIC «-//W3C//DTD HTML 4.01 Frameset//EN» «http://www.w3.org/TR/html4/frameset.dtd»>: HTML с фреймами.

2. HTML 5

<!DOCTYPE html>: для всех документов.

3. XHTML 1.0

<!DOCTYPE html PUBLIC «-//W3C//DTD XHTML 1.0 Strict//EN» «http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd»>: строгий синтаксис XHTML;

<!DOCTYPE html PUBLIC «-//W3C//DTD XHTML 1.0 Transitional//EN» «http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd»>: переходный синтаксис XHTML;

<!DOCTYPE html PUBLIC «-//W3C//DTD XHTML 1.0 Frameset//EN» «http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd»>: XHTML с фреймами.

4. XHTML 1.1

<!DOCTYPE html PUBLIC «-//W3C//DTD XHTML 1.1//EN» «http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd»>: для всех документов.

Для чего предназначен тег <head>?

Тег <head> предназначен для хранения других элементов, цель которых — помочь браузеру в ра-

боте с данными. Также внутри этого контейнера находятся метатеги, которые используются для хранения информации предназначенной для браузеров и поисковых систем. Например, механизмы поисковых систем обращаются к метатегам для получения описания сайта, ключевых слов и других данных.

Содержимое тега <head> не отображается напрямую на web-странице, за исключением тега <title> устанавливающего заголовок окна.

Внутри контейнера <head> допускается размещать следующие элементы: <base>, <basefont>, <bgsound>, <link>, <meta>, <script>, <style>, <title>.

Синтаксис:

```
<head>
  ...
</head>
```

Специфические атрибуты:

profile: указывает адрес профиля метаданных.

Чем отличается от ?

 - блочный, а - строчный элементы. Поэтому формирует блок из того, что в нем располагается с новой строки, а не переносит элементы, размещая их в строку. Так же стоит отметь, что согласно рекомендациям w3c линейный тег не может включать в себя блочные теги, поэтому обычно используется для разметки блоков, а - отрывков текста.

Как обозначаются комментарии в HTML?

Комментарий в HTML-коде задаётся так: **<!-- комментарий -->**

Комментарии можно использовать в любом месте страницы, кроме тега <title> — внутри него они не работают. Внутри тега <style> HTML-комментарии тоже не работают, так как в CSS код комментируется другим способом.

Каким образом задаётся адрес документа, на который следует перейти?

Для создания ссылок на другие документы используется тег <a>. В зависимости от присутствия атрибутов name или href тег <a> устанавливает ссылку или якорь. Якорем называется закладка внутри страницы, которую можно указать в качестве цели ссылки. При использовании ссылки, которая указывает на якорь, происходит переход к закладке внутри web-страницы.

Синтаксис:

```
<a href=»URL»>...</a>
<a name=»идентификатор»>...</a>
```

Специфические атрибуты:

- accesskey: активация ссылки с помощью комбинации клавиш;
- coords: устанавливает координаты активной области;

- download: предлагает скачать указанный по ссылке файл;
- href: задает адрес документа, на который следует перейти. Адрес ссылки может быть абсолютным и относительным. Абсолютные адреса работают везде и всюду независимо от имени сайта или веб-страницы, где прописана ссылка. Относительные ссылки, как следует из их названия, построены относительно текущего документа или корня сайта;
- hreflang: идентифицирует язык текста по ссылке;
- name: устанавливает имя якоря внутри документа;
- rel: отношения между ссылаемым и текущим документами;
- rev: отношения между текущим и ссылаемым документами;
- shape: задает форму активной области ссылки для изображений;
- tabindex: определяет последовательность перехода между ссылками при нажатии на кнопку Tab;
- target: имя окна или фрейма, куда браузер будет загружать документ;
- title: добавляет всплывающую подсказку к тексту ссылки;
- type: указывает MIME-тип документа, на который ведёт ссылка.

Как сделать ссылку на адрес электронной почты?

Создание ссылки на адрес электронной почты делается почти также, как и ссылка на web-страницу. Только вместо URL указывается mailto:»адрес электронной почты»

```
<a href=»mailto:user@address.net»>Напиши мне!</a>
```

Для чего предназначен тег ?

Тег предназначен для акцентирования текста. Браузеры отображают такой текст курсивным начертанием.

```
<em>Текст</em>
```

Для чего предназначены теги <ol>, <ul>, ?

Теги <ol>, <ul> и <li> предназначены для оформления списков.

: нумерованный список, т.е. каждый элемент списка начинается с числа или буквы и увеличивается по нарастающей.
: маркированный список, каждый элемент которого начинается с небольшого символа — маркера.
: отдельный элемент списка. Внешний тег <ul> или <ol> устанавливает тип списка — маркированный или нумерованный.

```
<ol>Нумерованый список
  <li>первый</li>
  <li>второй</li>
  <li>третий</li>
</ol>

<ul>Маркированный список
  <li>первый</li>
  <li>второй</li>
  <li>третий</li>
</ul>
```

Для чего предназначены теги <dl>, <dt>, <dd>?

Теги <dl>, <dt>, <dd> предназначены для создания списка определений.

Каждый такой список начинается с контейнера <dl>, куда входит тег <dt> создающий термин и тег <dd> задающий определение этого термина. Закрывающий тег </dd> не обязателен, поскольку следующий тег сообщает о завершении предыдущего элемента. Тем не менее, хорошим стилем является закрывать все теги.

```
<dl>Список определений
  <dt>Термин</dt>
  <dd>Определение</dd>
</dl>
```

Для чего предназначены теги <tr>, <th>, <td>?

<tr>: служит контейнером для создания строки таблицы. Каждая ячейка в пределах такой строки может задаваться с помощью тега <th> или <td>. <th>: предназначен для создания одной ячейки заголовка таблицы. <td>: предназначен для создания одной ячейки таблицы.

```
<table>
  <tr>
    <th>Заголовок</td>
  </tr>
  <tr>
    <td>Строка</td>
  </tr>
</table>
```

Обязательно ли писать атрибут alt в теге <img>?

Да, писать его обязательно.

Атрибут alt устанавливает альтернативный текст для изображений. Такой текст позволяет получить текстовую информацию о рисунке при отключенной в браузере загрузке изображений. Поскольку загрузка изображений происходит после получения браузером информации о нем, то замещающий рисунок текст появляется раньше. А уже по мере загрузки текст будет сменяться изображением.

```
<img src=»forest.jpg» alt=»Лес»>
```

В каком регистре лучше писать HTML-код?

Весь HTML-код рекомендуется писать в нижнем регистре: это относится к названиям элементов, названиям атрибутов, значениям атрибутов (кроме текста/CDATA), селекторам, свойствам и их значениям (кроме текста).

Не рекомендуется

```
<A HREF=»/»>Домой</A>
```

Рекомендуется

```
<img src=»forest.jpg» alt=»Лес»>
```

Что такое «мнемоника (entity)»?

Мнемоника (entity) - это конструкция из символа & и буквенного (или цифрового кода) после нее, предназначенная для замещения символов, которые запрещены для использования в HTML в «явном виде».

имеет мнемонику

Основы CSS

Что такое «CSS»?

CSS, Cascading Style Sheets (каскадные таблицы стилей) - формальный язык описания внешнего вида документа, написанного с использованием языка разметки, который применяется к элементам web-страницы для управления их видом и положением.

Основной целью разработки CSS являлось разделение описания логической структуры web-страницы, которое производится с помощью HTML или других языков разметки от описания внешнего вида этой web-страницы, которое производится с помощью CSS.

Как в CSS обозначаются комментарии?

Чтобы пометить, что текст является комментарием, применяют конструкцию /* ... */

Что такое «селектор»?

Селектор – это правило, на основании которого осуществляется выбор элементов в HTML документе для того, чтобы применить к ним определённые стили.

```
p {
text-align: center;
font-size: 20px;
}
/* p – это селектор, text-align и font-size – это свойства, а center и 20px – значения. */
```

Перечислите основные виды селекторов.

селектор * - выбор всех элементов;
селектор элемента - выбор всех элементов в HTML документе, имеющих указанный тег (например: div);
селектор класса - выбор всех элементов в HTML документе, имеющих указанный класс (например: .center);
селектор идентификатора - выбор элемента в HTML документе, имеющего указанный идентификатор (например: #footer);
селекторы псевдоклассов - выбор всех элементов в HTML документе, имеющих указанный псевдокласс (например: p:first-of-type);
селекторы атрибутов - выбор элементов в зависимости от указанного атрибута элемента или его значения (например: [href*=»youtube»]).

Что такое псевдокласс?

Псевдокласс определяет динамическое состояние элементов, которое изменяется из-за действий пользователя, или же соответствует текущему положению в дереве документа. В отличие от настоящего класса, в явном виде псеводкласс в HTML не указывается, а в CSS указывается через : непосредственно после селектора.

Наиболее известные псевдоклассы:

:link применяется к непосещенным ссылкам;
:visited применяется к посещенным ссылкам;
:hover применяется, когда курсор мыши находится в пределах элемента, но не активирует его;
:active применяется при активации элемента;
:focus применяется к элементу при получении им фокуса;

:first-child применяется к первому дочернему элементу селектора, который расположен в дереве элементов документа.

```
a.snowman:link {
  color: blue;
}
a.snowman:visited {
  color: purple;
}
a.snowman:active {
  color: red;
}
a.snowman:hover {
  text-decoration: none;
  color: blue;
  background-color: yellow;
}
```

Какие существуют селекторы аттрибутов?

[атрибут] - все элементы, имеющие указанный атрибут;
[атрибут=значение] - все элементы, имеющие атрибут, значение которого равно «значение»;
[атрибут^=занчение] - все элементы, имеющие атрибут, значение которого начинается с значение;
[атрибут|=значение] - все элементы, имеющие атрибут, значение которого равно значение или начинается с значение следующим образом значение-* (значение с обязательным дефисом, после которого идёт остальное содержимое значения);
[атрибут$=значение] - все элементы, имеющие атрибут, значение которого заканчивается на значение;
[атрибут*=значение] - все элементы, имеющие атрибут, значение которого содержит подстроку значение;
[атрибут~=значение] - все элементы, имеющие атрибут, значение которого содержит значение как одно из значений через пробел.

В чем разница между #my и .my?

#my — селектор идентификатора, а .my — селектор класса.

В чем разница между margin и padding?

margin — внешний отступ, а padding — внутренний отступ.

В чем заключается разница между значениями 0 и auto в свойстве margin?

В вертикальных полях — auto всегда означает 0. В горизонтальных полях — auto означает 0 только тогда, когда свойство width также auto.

Какое свойство задает цвет фона?

Цвет фона задает свойство background-color.

Как убрать подчеркивание для всех ссылок на странице?

```
a {
  text-decoration: none;
}
```

Для чего используется свойство clear?

clear устанавливает, с какой стороны элемента запрещено его обтекание другими элементами.

Как сделать жирным текст во всех элементах <p>?

```
p {
  font-weight: bold;
}
```

Как задать красный цвет для всех элементов, имеющих класс red?

```
.red {
  color: red;
}
```

Основы WEB

Что такое WWW?

WWW, World Wide Web (Всемирная паутина) — распределённая система, предоставляющая доступ к связанным между собой документам, расположенным на различных компьютерах, подключённых к Интернету. Для обозначения этого термина также используют слово web.

Что такое W3C?

W3C, World Wide Web Consortium (Консорциум Всемирной паутины) — организация, разрабатывающая и внедряющая технологические стандарты для WWW.

W3C разрабатывает для Интернета единые принципы и стандарты, называемые «рекомендациями» (W3C Recommendations), которые затем внедряются производителями программ и оборудования. Таким образом достигается совместимость между программными продуктами и аппаратурой различных компаний.

Какие существуют уровни модели OSI?

#	Уровень (layer)	Тип данных (PDU)	Функции	Примеры
7	Прикладной (application)	-	Доступ к сетевым службам	HTTP, FTP
6	Представительский (presentation)	-	Представление и шифрование данных	ASCII, JPEG
5	Сеансовый- (session)	-	Управление сеансом связи	RPC, PAP
4	Транспортный (transport)	Сегменты (segment)/ Дейтаграммы (datagram)	Прямая связь между конечными пунктами и надёжность	TCP, UDP
3	Сетевой (network)	Пакеты (packet)	Определение маршрута и логическая адресация	IP, AppleTalk
2	Канальный (data link)	Биты (bit)/Кадры (frame)	Физическая адресация	Ethernet, IEEE 802.2, L2TP
1	Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными данными	USB, витая пара

Что такое TCP/IP?

TCP/IP - это два основных сетевых протокола Internet. Часто это название используют и для обозначения сетей, работающих на их основе.

IP (Internet Protocol) - маршрутизируемый протокол, отвечающий за IP-адресацию, маршрутизацию, фрагментацию и восстановление пакетов. В его задачу входит продвижение пакета между сетями – от одного маршрутизатора до другого и тех пор, пока пакет не попадет в сеть назначения. В отличие от протоколов прикладного и транспортного уровней, протокол IP разворачивается не только на хостах, но и на всех шлюзах (маршрутизаторах). Этот протокол работает без установления соединения и без гарантированной доставки.

В настоящее время используются следующие две версии протокола IP:

IPv6 — IP-адрес имеет разрядность 128 бит и записывается в виде восьми 16-битных полей, с использованием шестнадцатеричной системы счисления и с возможностью сокращения двух и более последовательных нулевых полей до ::, например: 2001:db8:42::1337:cafe

IPv4 — IP-адрес имеет разрядность 32 бита и записывается в виде четырех десятичных чисел в диапазоне 0 … 255 через точку, например: 192.0.2.34.

TCP (Transfer Control Protocol) - протокол, обеспечивающий надежную, требующую логического соединения связь между двумя компьютерами. Отвечает за установление соединения, упорядочивание посылаемых пакетов и восстановление пакетов, потерянных в процессе передачи.

Стек протоколов TCP/IP включает в себя четыре уровня:

- канальный уровень (link layer) - например Ethernet, IEEE 802.11 Wireless Ethernet, физическая среда и принципы кодирования информации
- сетевой уровень (Internet layer) - например IP
- транспортный уровень (transport layer) - например TCP, UDP
- прикладной уровень (application layer) - например HTTP, FTP, DNS

TCP-соединение двух узлов начинается с handshake (рукопожатия):

- Узел A посылает узлу B специальный пакет SYN — приглашение к соединению
- B отвечает пакетом SYN-ACK — согласием об установлении соединения
- A посылает пакет ACK — подтверждение, что согласие получено

После этого TCP соединение считается установленным и приложения, работающие в этих узлах, могут посылать друг другу пакеты с данными.

В заголовке TCP/IP пакета указывается:

IP-адрес отправителя
IP-адрес получателя
Номер порта

Что такое UDP?

UDP, User Datagram Protocol (Протокол пользовательских датаграмм) — протокол, который обеспечивает доставку без требований соединения с удаленным модулем UDP и обязательного подтверждения получения.

К заголовку IP-пакета UDP добавляет всего четыре поля по 2 байта каждое:

- поле порта источника (source port)
- поле порта пункта назначения (destination port)
- поле длины (length)
- поле контрольной суммы (checksum)

Поля «порт источника» и «контрольная сумма» не являются обязательными для использования в IPv4. В IPv6 необязательно только поле «порт отправителя».

UDP используется DNS, SNMP, DHCP и другими приложениями.

Чем отличаются TCP и UDP?

TCP — ориентированный на соединение протокол, что означает необходимость «рукопожатия» для установки соединения между двумя хостами. Как только соединение установлено, пользователи могут отправлять данные в обоих направлениях.

Надёжность — TCP управляет подтверждением, повторной передачей и тайм-аутом сообщений. Производятся многочисленные попытки доставить сообщение. Если оно потеряется на пути, сервер вновь запросит потерянную часть. В TCP нет ни пропавших данных, ни (в случае многочисленных тайм-аутов) разорванных соединений.

Упорядоченность — если два сообщения последовательно отправлены, первое сообщение достигнет приложения-получателя первым. Если участки данных приходят в неверном порядке, TCP отправляет неупорядоченные данные в буфер до тех пор, пока все данные не могут быть упорядочены и переданы приложению.

Тяжеловесность — TCP необходимо три пакета для установки соединения перед тем, как отправить данные. TCP следит за надёжностью и перегрузками.

Потоковость — данные читаются как поток байтов, не передается никаких особых обозначений для границ сообщения или сегментов.

UDP — более простой, основанный на сообщениях протокол без установления соединения. Протоколы такого типа не устанавливают выделенного соединения между двумя хостами. Связь достигается путём передачи информации в одном направлении от источника к получателю без проверки готовности или состояния получателя.

Ненадёжность — когда сообщение посылается, неизвестно, достигнет ли оно своего назначения — оно может потеряться по пути. Нет таких понятий как подтверждение, повторная передача, тайм-аут.

Неупорядоченность — если два сообщения отправлены одному получателю, то порядок их достижения цели не может быть предугадан.

Легковесность — никакого упорядочивания сообщений, никакого отслеживания соединений и т. д. Это лишь транспортный уровень.

Датаграммы — пакеты посылаются по отдельности и проверяются на целостность только если они прибыли. Пакеты имеют определенные границы, которые соблюдаются после получения, то есть операция чтения на получателе выдаст сообщение таким, каким оно было изначально послано.

Отсутствие контроля перегрузок — для приложений с большой пропускной способностью существует шанс вызвать коллапс перегрузок, если только они не реализуют меры контроля на прикладном уровне.

Что такое протокол передачи данных? Какие протоколы вы знаете?

Протокол передачи данных — набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами. Эти соглашения задают единообразный способ передачи сообщений и обработки ошибок при взаимодействии программного обеспечения разнесённой в пространстве аппаратуры, соединённой тем или иным интерфейсом.

Наиболее известные протоколы передачи данных:

HTTP (Hyper Text Transfer Protocol)
FTP (File Transfer Protocol)
POP3 (Post Office Protocol)
SMTP (Simple Mail Transfer Protocol)
TELNET (TErminaL NETwork)

Что такое HTTP и HTTPS? Чем они отличаются?

HTTP, HyperText Transfer Protocol (Протокол передачи гипертекста) — протокол прикладного уровня передачи данных.

Основой HTTP является технология «клиент-сервер»:

Потребители (клиенты), которые инициируют соединение и посылают запрос;
Поставщики (серверы), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.

Для идентификации ресурсов HTTP использует глобальные URI.

HTTP не сохраняет своего состояния. Это означает отсутствие сохранения промежуточного состояния между парами «запрос-ответ».

Структура протокола:

Стартовая строка (starting line) — определяет тип сообщения;
Заголовки (headers) — характеризуют тело сообщения, параметры передачи и прочие сведения;
Тело сообщения (message body) — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой.

Заголовки и тело сообщения могут отсутствовать, но стартовая строка является обязательным элементом, так как указывает на тип запроса/ответа.

HTTPS, HyperText Transfer Protocol Secure — расширение протокола HTTP, поддерживающее шифрование. Данные, передаваемые по протоколу HTTPS, «упаковываются» в криптографический протокол SSL или TLS, что обеспечивает защиту от атак, основанных на прослушивании сетевого соединения (при условии, что будут использоваться шифрующие средства и сертификат сервера проверен и ему доверяют).

Различия HTTP и HTTPS:

HTTPS является расширением HTTP.

HTTP использует не зашифрованное соединение. Соединение по HTTPS поддерживает шифрование.

Работа по HTTP быстрей и менее ресурсоёмко, т.к. шифрование HTTPS требует дополнительных затрат.

Порты по умолчанию: в случае HTTP это TCP-порт 80, для HTTPS - TCP-порт 443.

Что такое FTP?

FTP, File Transfer Protocol (Протокол передачи файлов) — протокол передачи файлов между компьютерами в сети TCP. С его помощью можно подключаться к FTP-серверам, просматривать содержимое их каталогов и загружать файлы с сервера или на сервер. Протокол построен на архитектуре «клиент-сервер» и использует разные сетевые соединения для передачи команд и данных между клиентом и сервером.

По умолчанию использует TCP-порт 21.

Чем отличаются методы GET и POST?

GET передает данные серверу используя URL, тогда как POST передает данные, используя тело HTTP запроса. Длина URL ограничена 1024 символами, это и будет верхним ограничением для данных, которые можно отослать через GET. POST может отправлять гораздо большие объемы данных. Лимит устанавливается web-server и составляет обычно около 2 Mb.

Передача данных методом POST более безопасна, чем методом GET, так как секретные данные (например пароль) не отображаются напрямую в web-клиенте пользователя, в отличии от URL, который виден почти всегда. Иногда это преимущество превращается в недостатком - вы не сможете послать данные за кого-то другого.

Что такое MIME тип?

MIME, Multipurpose Internet Mail Extension (Многоцелевые расширения Интернет-почты) — спецификация для передачи по сети файлов различного типа: изображений, музыки, текстов, видео, архивов и др. В HTML указание MIME-типа используется при передаче данных форм и вставки на страницу различных объектов.

Что такое Web server?

Web server (Веб-сервер) — сервер, принимающий HTTP-запросы от клиентов и выдающий им HTTP-ответы. Так называют как программное обеспечение, выполняющее функции web-сервера, так и непосредственно компьютер, на котором это программное обеспечение работает.

Web-серверы могут иметь различные дополнительные функции, например:

- автоматизация работы web-страниц;
- ведение журнала обращений пользователей к ресурсам;
- аутентификация и авторизация пользователей;
- поддержка динамически генерируемых страниц;
- поддержка HTTPS для защищённых соединений с клиентами.

Наиболее известные web-серверы:

Apache
Microsoft IIS
nginx

Что такое Web application?

Web application (Веб-приложение) - клиент-серверное приложение, в котором клиентом выступает браузер, а сервером — web-сервер. Логика web application распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, а обмен информацией происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому web application является кроссплатформенным сервисом.

Что такое Application server?

Application Server (Сервер приложений) — программа, представляющая собой сервер, который занимается системной поддержкой приложений и обеспечивает их жизненный цикл в соответствии с правилами, определёнными в спецификациях. Может работать как полноценный самостоятельный

web-сервер или быть поставщиком страниц для другого web-сервера. Обеспечивает обмен данными между приложениями и клиентами, берёт на себя выполнение таких функций, как создание программной среды для функционирующего приложения, идентификацию и авторизацию клиентов, организацию сессии для каждого из них.

Наиболее известные серверы приложений Java:

Apache Tomcat
Jetty
JBoss
GlassFish
IBM WebSphere
Oracle Weblogic

Чем отличаются Web server и Application server?

Понятие web server относится скорее к способу передачи данных (конкретно, по протоколу HTTP), в то время как понятие Application server относится к способу выполнения этих самых приложений (конкретно, удаленная обработка запросов клиентов при помощи каких-то программ, размещенных на сервере). Эти понятия нельзя ставить в один ряд. Они обозначают разные признаки программы. Какие-то программы удовлетворяют только одному признаку, какие-то - нескольким сразу.

Apache Tomcat умеет выполнять приложения? Да, значит он является application server. Apache Tomcat умеет отдавать данные по HTTP? - Да. Следовательно он является web server.

Возьмите какую-нибудь базу данных, в которой на хранимых процедурах описана сложная логика и можно в ответ на SQL-запросы отправлять даже sms. Такую базу данных можно назвать application server, но web server - уже нет, потому что все это не работает с клиентом по HTTP протоколу.

Возьмите чистый Apache, в котором не включены никакие модули для поддержки языков программирования. Он умеет отдавать только статичные файлы и картинки по протоколу HTTP. Это web server, но не application server. Включите модуль для поддержки PHP и разместите там программу на PHP, которая делает запросы к базе данных и динамически формирует страницы. Теперь Apache стал и application server.

Что такое AJAX?
Как принципиально устроена эта технология?

AJAX, Asynchronous Javascript and XML (Асинхронный Javascript и XML) — подход к построению интерактивных пользовательских интерфейсов web-приложений, заключающийся в «фоновом» обмене данными браузера и web-сервера. В результате, при обновлении данных web-страница не перезагружается полностью и web-приложения становятся быстрее и удобнее.

При использовании AJAX:

Пользователь заходит на web-страницу и взаимодействует с каким-нибудь её элементом.
Скрипт на языке JavaScript определяет, какая информация необходима для обновления страницы.
Браузер отправляет соответствующий запрос на web-сервер.
Web-сервер возвращает только ту часть документа, на которую пришёл запрос.
Скрипт вносит изменения с учётом полученной информации (без полной перезагрузки страницы).

AJAX базируется на двух основных принципах:

1. использование технологии динамического обращения к серверу «на лету» (без перезагрузки страницы полностью) через динамическое создание:
   - дочерних фреймов;
   - тега <script>;
   - тега <img>.
2. использование DHTML для динамического изменения содержания страницы;

AJAX не является самостоятельной технологией, это концепция использования нескольких смежных технологий:

(X)HTML, CSS для подачи и стилизации информации;
DOM-модель, операции над которой производятся Javascript на стороне клиента, для обеспечения динамического отображения и взаимодействия с информацией;
XMLHttpRequest или другой транспорт (IFrame, SCRIPT-тег, ...) для асинхронного обмена данными с web-сервером;
JSON или любой другой подходящий формат (форматированный HTML, текст, XML, ...) для обмена данными.

Что такое WebSocket?

WebSocket — протокол полнодуплексной связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и web-сервером в режиме реального времени.

Протокол WebSocket определяет две URI схемы

ws: - нешифрованное соединение
wss: - шифрованное соединение

Что такое JSON?

JSON, JavaScript Object Notation — текстовый формат обмена данными, основанный на JavaScript.

JSON представляет собой (в закодированном виде) одну из двух структур:

Набор пар «ключ:значение»;
Упорядоченный набор значений.

Ключом может быть только строка (регистрозависимая: имена с буквами в разных регистрах считаются разными).

В качестве значений могут быть использованы:

Объект — неупорядоченное множество пар «ключ:значение», заключённое в фигурные скобки { }. Ключ описывается строкой, между ним и значением стоит символ :. Пары ключ-значение отделяются друг от друга запятыми;
Массив (одномерный) — упорядоченное множество значений. Массив заключается в квадратные скобки [ ]. Значения разделяются запятыми.
Число;
Литералы true, false и null;
Строка — упорядоченное множество из нуля или более символов Unicode, заключенное в кавычки « «. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты \, или записаны шестнадцатеричным кодом в кодировке UTF-8 в виде \ uFFFF.

Что такое JSON схема?

JSON Schema — один из языков описания структуры JSON-документа, используя синтаксис JSON. Это самоописательный язык: при его использовании для обработки данных и описания их допустимости могут использоваться одни и те же инструменты сериализации/десериализации.

Что такое cookies?

Cookies («куки») — небольшой фрагмент данных, отправленный web-сервером и хранимый на устройстве пользователя. Всякий раз при попытке открыть страницу сайта, web-клиент пересылает соответствующие этому сайту cookies web-серверу в составе HTTP-запроса. Применяется для сохранения данных на стороне пользователя и на практике обычно используется для:

- аутентификации пользователя;
- хранения персональных предпочтений и настроек пользователя;
- отслеживания состояния сеанса доступа пользователя;
- ведения разнообразной статистики.

Что такое «сессия»?

Сессия – промежуток времени между первым и последним запросами, которые пользователь отправляет со своего устройства на сервер сайта. Завершается сессия в случае, если со стороны пользователя не поступало запросов в течение определенного промежутка времени или же при обрыве связи.

Что такое «авторизация» и «аутентификация»? Чем они отличаются?

Аутентификация - это проверка соответствия субъекта и того, за кого он пытается себя выдать, с помощью некой уникальной информации (отпечатки пальцев, цвет радужки, голос и тд.), в простейшем случае - с помощью имени входа и пароля.

Авторизация - это проверка и определение полномочий на выполнение некоторых действий (например, чтение файла) в соответствии с ранее выполненной аутентификацией.

Очевидно, что это разные понятия, но при этом без первого не может быть второго и наоборот. То есть имея разрешение на работу, вы не сможете оказаться на рабочем месте без предъявления пропуска, равно как и нет смысла в демонстрации пропуска, если вы не планируете работать. Именно тот факт, что одного не бывает без другого, и вызывает у людей заблуждение, что это одно и то же.