Оглавление

1 **Core-1**

ООП

2	**Что такое ООП**	**ООП** - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Согласно парадигме ООП программа состоит из обьектов, обменивающихся сообщениями. Обьекты могут обладать состоянием, единственный способ изменить состояние обьекта - передать ему сообщение, в ответ на которое, обьект может изменить собственное состояние. Класс — это описание еще не созданного объекта, как бы общий шаблон, состоящий из полей, методов и конструктора, а объект – экземпляр класса, созданный на основе этого описания.
3	**Какие преимущества у ООП**	**Легко читается** - не нужно выискивать в коде функции и выяснять, за что они отвечают
4		**Быстро пишется -** можно быстро создать сущности, с которыми должна работать программа.
5		**Простота реализации большого функционала** - т.к. на написание кода уходит меньше времени, можно гораздо быстрее создать приложение с множеством возможностей
6		**Меньше повторений кода** - не нужно писать однотипные функции для разных сущностей
7	**Какие недостатки у ООП**	
8		**Снижает производительность** - многие вещи технически реализованы иначе, поэтому они используют больше ресурсов.
9		**Сложно начать** - парадигма ООП сложнее функционального программирования, поэтому на старт уходит больше времени
10	**Назовите основные принципы ООП**	**Инкапсуляция** **Наследование** **Полиморфизм**
11	**Что такое инкапсуляция? (С примером)**	Свойство системы, которое объединяет данные и методы, манипулирующие этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. Инкапсуляция - это объединение данных и методов работы с этими данными в одной упаковке («капсуле»). Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса
12	**Что такое наследование? (С примером)**	**Свойство** системы, которое **позволяет описать** новый **класс на основе уже существующего** с частично или полностью заимствованной функциональностью.
13	**Что такое полиморфизм? (С примером)**	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). **Полиморфизм бывает динамическим (переопределение) и статическим (перегрузка).** 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); -**параметрическая** функция ведет себя одинаково для аргументов разных типов (например, функция add() — одинаково кладет в контейнер элементы разных типов).
14	**Что такое ассоциация**	Есть два типа связи между объектами: ассоциация, которая делится на композицию и агрегацию, и наследование. Ассоциация - обозначает связь между обьектами. Например, игрок играет в определенной команде. Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом. Например Менеджер может выписать Счет. Соответственно возникает ассоциация между Менеджером и Счетом. Еще пример — Преподаватель и Студент — т.е. какой-то Студент учится у какого-то Преподавателя. Ассоциация и есть описание связи между двумя объектами. Студент учится у Преподавателя. Идея достаточно простая — два объекта могут быть связаны между собой и это надо как-то описать. http://java-course.ru/begin/relations/
15		**class Team** **{** **}** **class Player** **{** **public Team Team { get; set; }** **}**
16	**Что такое композиция**	**Композиция** — еще более «жесткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то. Например Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно. В отличии от студента, который может входить и в другие группы тоже. Например, в класс автомобиля содержит объект класса электрического двигателя:

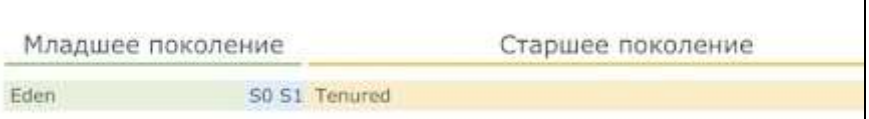
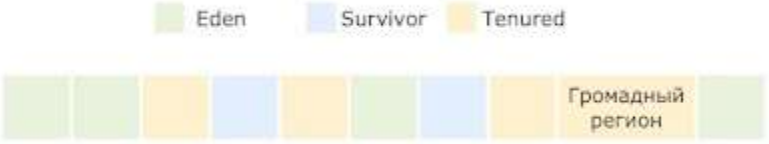
17		**public** class ElectricEngine{ }
		
		public class Car {
		ElectricEngine engine;
		public Car()
		{
		engine = **new** ElectricEngine();
		}

		}
18		При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.
19	**Что такое агрегация**	Агрегация определяет отношение **HAS A**, но связь слабее чем в композиции, т.к. обьекты равноправны.
20	**Расскажите про раннее и позднее связывание.**	**Связывание есть наличие связи между вызываемым методом программы и написанным кодом.** **Ранее свзяывание** Если метод известен компилятору, то происходит ранее связывание на этапе компиляции (early binding), также называют статическим связыванием. **Позднее связывание** (late binding) - вызов метода возможен только во время выполнения, т.к. у компилятора нет информации, чтобы проверить корректность такого вызова. В java это возможно при помощи рефлексии. **Статическое связывание** используется для final, перегруженных, приватных, статических методов, в то время как динамическое связывание используется для разрешения переопределенных методов. Все абстрактные методы разрешаются при помощи динамического связывания. В случае статического связывания используются не конкретные объекты, а информация о типе, то есть используется тип ссылочной переменной. С другой стороны, при динамическом связывании для нахождения нужного метода используется конкретный объект.
21	**SOLID**	**SOLID** — это акроним, образованный из заглавных букв первых пяти принципов ООП и проектирования. **S(Single Responsibility Principle)** - принцип единственной ответственности - каждый класс выполняет лишь одну задачу. Легкая модификация в будущем, простое тестирование, класс не имеет зависимостей на другие классы. **O(Open Closed Principle)** - принцип открытости/закрытости - программные сущности открыты для расширения и закрыты для модификации. Чтобы не сломать логику в классе-родителе, мы унаследуемся от него и реализуем что-то своё, и используем свой класс. **L(Liskov's Substitution Principle)** - принцип подстановки барбары лисков - объекты в программе можно заменить их наследниками без изменения свойств программы. **I(Interface Segregation Principle)** - принцип разделения интерфейса - много специализированных интерфейсов лучше, чем один общий **D(Dependency Inversion Principle)** - принцип инверсии зависимостей - зависимость на абстракциях. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

		Использование: Создание интерфейсов и их реализаций. Пример: терминал оплаты(абстракция) и разные карты оплаты.
	Java	
23	**Какая основная идея языка?**	**«Написано однажды - работает везде»**. Идея основывается в написании одного кода, который будет работать на любой платформе.
24	**За счет чего обеспечивается кроссплатформенность?**	Кроссплатформенность была достигнута **за счёт** создания **виртуальной машина Java**. Java Virtual Machine или JVM - это программа, являющаяся **прослойкой между операционной системой** и **Java программой**. В среде виртуальной машины выполняются коды Java программ. Сама JVM реализована для разных ОС. Что байт код для JVM может исполняться везде где установлена JVM. Код не нужно перекомпилировать под каждую из платформ.
25	**Какие преимущества у java?**	**Объектно-ориентированное программирование** - структура данных становится объектом, которым можно управлять для создания отношений между различными объектами. **Язык высокого уровня с простым синтаксисом и плавной кривой обучения** - синтаксис Java основан на C ++, поэтому Java похожа на C. Тем не менее, синтаксис Java проще, что позволяет новичкам быстрее учиться и эффективнее использовать код для достижения конкретных результатов. **Стандарт для корпоративных вычислительных систем** - корпоративные приложения — главное преимущество Java с 90-х годов, когда организации начали искать надежные инструменты программирования не на C. **Безопасность** - благодарю отсутсвию указателей и Security Manager (политика безопасности, в которой можно указать правила доступа, позволяет запускать приложения Java в "песочнице"). **Независимость от платформы** - Можно создать Java-приложение на Windows, скомпилировать его в байт-код и запустить его на любой другой платформе, поддерживающей виртуальную машину Java (JVM). Таким образом, JVM служит уровнем абстракции между кодом и оборудованием. **Язык для распределенного программирования и комфортной удаленной совместной работы** - Специфическая для Java методология распределенных вычислений называется Remote Method Invocation (RMI). RMI позволяет использовать все преимущества Java: безопасность, независимость от платформы и объектно-ориентированное программирование для распределенных вычислений. Кроме того, Java также поддерживает программирование сокетов и методологию распределения CORBA для обмена объектами между программами, написанными на разных языках. **Автоматическое управление памятью** Разработчикам Java не нужно вручную писать код для управления памятью благодаря автоматическому управлению памятью (AMM). **Многопоточность** Поток — наименьшая единица обработки в программировании. Чтобы максимально эффективно использовать время процессора, Java позволяет запускать потоки одновременно, что называется многопоточностью. **Стабильность и сообщество** Сообщество разработчиков Java не имеет себе равных. Около 45% респондентов опроса StackOverflow 2018 используют Java.
26	**Какие недостатки у java?**	**Платное коммерческое использование (с 2019)** **Низкая производительность** из-за компиляции и абстракции с помощью виртуальной машины, а также приложение очистки памяти.

		Не развитые инструменты по созданию GUI приложений на чистой java. **Многословный код** Java — это более легкая версия неприступного C ++, которая вынуждает программистов прописывать свои действия словами из английского языка. Это делает язык более понятным для неспециалистов, но менее компактным.
27	**Что такое JDK? Что в него входит?**	**JDK (Java Development Kit)** - **включает JRE и набор инструментов разработчика** приложений на языке Java: - компилятор Java (javac) - стандартные библиотеки классов java - примеры - документацию - различные утилиты
28	**Что такое JRE? Что в него входит?**	**JRE (java Runtime Environment)** - минимально-необходимая реализация виртуальной машины для исполнения Java-приложений. **Состоит из JVM, ClassLoader и** стандартного **набора библиотек** и классов Java
29	**Что такое JVM?**	**JVM (Java Virtual Machine)** - виртуальная машина Java исполняет байт-код Java, предварительно созданный из кода JIT компилятором, с помощью встроенного интерпретатора байткода. **HotSpot** представляет собой реализацию концепции JVM.
30	**Что такое byte code?**	Байт-код Java — набор инструкций, скомпилированный компилятором, исполняемый JVM.
31	**Что такое загрузчик классов (classloader)?**	Используется **для передачи в JVM скомпилированного байт-кода**, **хранится в** файлах с расширением **.class** При запуске JVM, используются три загрузчика классов: **- Bootstrap ClassLoader - базовый загрузчик** - загружает платформенные классы JDK из архива rt.jar **- AppClassLoader - системный загрузчик** - загружает классы приложения, определенные в CLASSPATH **- Extension ClassLoader - загрузчик расширений** - загружает классы расширений, которые по умолчанию находятся в каталоге jre/lib/ext. ClassLoader 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. Каждый загрузчик хранит указатель на родительский, чтобы суметь передать загрузку если сам будет не в состоянии этого сделать.
32	**Что такое JIT?**	JIT (Just-in-time compilation) - компиляция на лету или динамическая компиляция - технология увеличения производительности программных систем, использующих байт-код, путем компиляции байт-кода в машинный код во время работы программы. В основном отвечает за оптимизацию производительности

		приложений во время выполнения. https://javahelp.online/osnovy/voprosy-otvety-sobesedovanie-java (Q13)
33	Что такое сборщик мусора? (Garbage collector)	Про 4 типа сборщиков мусора читай здесь: habr.com/ru/post/269621
34		Сборщик мусора выполняет две задачи: **- поиск мусора;** **- очистка мусора.** **Для обнаружения мусора есть два подхода:**
35		**- Учет ссылок (Reference counting);**
36		Учет ссылок - если обьект не имеет ссылок, он считается мусором. Проблема - не возможность выявить циклические ссылки, когда два обьекта не имеют внешних ссылок, но ссылаются друг на друга -> утечка памяти
37		**- Трассировка (Tracing).** (используется в HotSpot)6
38		Трассировка - до обьекта можно добраться из Корневых точке (GC root). До чего добраться нельзя - мусор. Всё, что доступно из «живого» объекта, также является «живым».
39		**Типы корневых точек (GC Roots) java приложения:**
40		- объекты в статических полях классов - объекты, доступные из стека потоков - объекты из JNI(java native interface) ссылок в native методах
41		**Процессы сборки мусора разделяются несколько видов:** **minor GC (малая)** - частый и быстрый, работает только с областью памяти "young generation"; - приложение приостанавливается на начало сборки мусора (такие остановки называются stop-the-world); - «живые» объекты из Eden перемещаются в область памяти «To»; - «живые» объекты из «From» перемещаются в «To» или в «old generation», если они достаточно «старые»; - Eden и «From» очищаются от мусора; - «To» и «From» меняются местами; - приложение возобновляет работу. **major GC (старшая)** - редкий и более длительный, затрагивает объекты старшего поколения. В принцип работы «major GC» добавляется процедура «уплотнения», позволяющая более эффективно использовать память. В процедуре живые объекты перемещаются в начало. Таким образом, мусор остается в конце памяти. **full GC (полная)** - полный сборщик мусора сначала запускает Minor, а затем Major (хотя порядок может быть изменен, если старое поколение заполнено, и в этом случае он освобождается первым, чтобы позволить ему получать объекты от молодого поколения).
42	Виды ссылок в Java	1) **StrongReference** — это самые обычные ссылки которые мы создаем каждый день, любая переменная ссылочного типа. StringBuilder builder = new StringBuilder(); - builder это и есть strong-ссылка на объект StringBuilder.
43		2) **SoftReference** — GC гарантировано удалит с кучи все объекты, доступные только по soft-ссылке, перед тем как бросит OutOfMemoryError. SoftReference это наш механизм кэширования объектов в памяти, но в критической ситуации, когда закончится доступная память, GC удалит не использующиеся объекты из памяти и тем самым попробует спасти JVM от завершения работы.

		StringBuilder builder = new StringBuilder(); SoftReference<StringBuilder> softBuilder = new SoftReference(builder); softBuilder.get() — вернет strong-ссылку на объект StringBuilder в случае если GC не удалил этот объект из памяти. В другом случае вернется null. softBuilder.clear() — удалит ссылку на объект StringBuilder То же самое работает для **WeakReference**.
44		3) **WeakReference** — если GC видит, что объект доступен только через цепочку weak-ссылок (исчезнули strong-ссылки), то он удалит его из памяти.
45		4) **PhantomReference** — если GC видит что объект доступен только через цепочку phantom-ссылок, то он его удалит из памяти. После нескольких запусков GC. Особенностей у этого типа ссылок две. Первая это то, что метод get() всегда возвращает null. Именно из-за этого PhantomReference имеет смысл использовать только вместе с ReferenceQueue. Вторая особенность – в отличие от SoftReference и WeakReference, GC добавит phantom-ссылку в ReferenceQueue **после** того как выполниться метод finalize().
46		**So in brief: Soft references try to keep the reference. Weak references don't try to keep the reference. Phantom references don't free the reference until cleared.**
47		**ReferenceQueue**. Он позволяет отслеживать момент, когда GC определит что объект более не нужен и его можно удалить. Именно сюда попадает Reference объект после того как объект на который он ссылается удален из памяти. При создании Reference мы можем передать в конструктор ReferenceQueue, в который будут помещаться ссылки после удаления.
48	**Stack и Heap**	Память процесса делится на Stack (стек) и Heap (куча) : - Stack содержит stack frame'ы, они делятся на три части: параметры метода, указатель на предыдущий фрейм и локальные переменные. - Структура Heap зависит от выбранного сборщика мусора. Читай про GC! MetaSpace - специальное пространство кучи, отделенное от кучи основной памяти. JVM хранит здесь весь статический контент. Это включает в себя все статические методы, примитивные переменные и ссылки на статические объекты. Кроме того, он содержит данные о байт-коде, именах и JIT-информации . До Java 7 String Pool также был частью этой памяти. Вкратце, при Serial/Parallel/CMS GC будет следующая структура:
49		
50		А при G1 GC:
51		

52		С помощью опций Xms и Xmx можно настроить начальный и максимально допустимый размер кучи соответственно. Существуют опции для настройки величины стека. - Heap - используется всем приложением, Stack - одним потоком исполняемой программы. - Новый обьект создается в heap, в stack размещается ссылка на него. В стеке размещаются локальные переменные примитивных типов. - Обьекты в куче доступны из любого места программы, стековая память не доступна для других потоков. - Если память стека закончилась JRE вызовет исключение StackOverflowError, если куча заполнена OutOfMemoryError - Размер памяти стека, меньше памяти кучи. Стековая память быстрее памяти кучи. - В куче есть ссылки между объектами и их классами. На этом основана рефлексия. Обе области хранятся в RAM.
	Процедурная Java	
54	**Какие примитивные типы данных есть в Java?**	Вещественные, целочисленные, логические и строковые. byte 8 бит от -128 до 127 short 16 бит от -32768 до 32767 int 32 бит от -2147483648 до 2147483647 long 64 бит от -9223372036854775808L до 9223372036854775807L float 32 от 1.4e-45f до 3.4e+38f double 64 от 4.9e-324 до 1.7e+308 char boolean
55	**Что такое char?**	**16-разрядное беззнаковое целое**, представляющее собой символ UTF-16 (буквы и цифры)
56	**Сколько памяти занимает boolean?**	**Зависит от реализации JVM** В стандартной реализации Sun JVM и Oracle HotSpot JVM тип boolean занимает 4 байта (32 бита), как и тип int. Однако, в определенных версия JVM имеются реализации, где в массиве boolean каждое значение занимает по 1-му биту.
57	**Что такое классы-обертки?**	Обертка — это специальный **класс**, **который хранит внутри** себя **значение примитива(объекты классов-оберток являются неизменяемыми (Immutable))**. Нужны для реализации дженериков.
58	**Что такое автоупаковка и автораспаковка?**	**Автоупаковка** - Преобразования примитивных типов в эквивалентные объекты **Автораспаковка** - процесс преобразования объектов в соответствующие им примитивные типы для присваивания ссылок-примитивов объектам их классов-оберток (и наоборот) не требуется ничего делать, все происходит автоматически. Для того, чтобы иметь возможность оперировать с простыми числами (и boolean) как с объектами были придуманы классы-обёртки.
59	**Что такое явное и неявное приведение типов? В каких случаях в java**	**Неявное приведение** – автоматическое **расширение** типа переменной **от меньшего к большему.** **Явное приведение** - **явное сужение от большего к меньшему.** Необходимо явно указать сужаемый тип.

	нужно использовать явное приведение?	В случае с объектами мы можем делать неявное(автоматическое) приведение от наследника к родителю, но не наоборот, иначе получим ClassCastException.
60	**Что такое пул интов?**	В Java есть пул(pool) целых чисел в промежутке [-128;127], так как это самый часто вречающийся диапазон. Т.е. если мы создаем Integer в этом промежутке, то вместо того, чтобы каждый раз создавать новый объект, JVM берет их из пула. Изменить размер кэша в HotSpot вы можете, указав ключ -XX:AutoBoxCacheMax=<размер>
61	**Какие нюансы у строк в Java?**	Класс **String** в Java - **неизменяемый** из-за модификатора final и отсутствия сеттера. Это нужно для реализации пула стрингов. **При редактировании** будет создаваться **новая строка**. **При копировании** новая строка не создается, а создается **ссылка на существующую** строку.
62	**Что такое пул строк?**	**Область памяти где хранятся обьекты строк**. **При создании** в пуле идет **поиск строки:** **-если НЕ находит - создается строка, возращается ссылка** -**если находит - возращает ссылку найденной** строки. При этом использование оператора new заставляет класс String создать новый объект, даже если такая строка уже есть в пуле. После этого можем использовать метод intern(), чтобы поместить этот объект в пул строк. Пул строк и Integer хранится в heap, но ссылки на объекты хранятся в stack.
63	**Почему не рекомендуется изменять строки в цикле? Что рекомендуется использовать?**	Т.к. **строка неизменяемый класс**, потребление ресурсов при редактировании, т.к. **каждую итерацию** при редактировании **будет создаваться новый** **обьект** строки. **Рекомендуется** использовать **StringBuilder или StringBuffer**.
64	**Почему строки не рекомендуется использовать для хранения паролей?**	**1. Пул строк** Так как строки в Java хранятся в пуле строк, то ваш пароль в виде обычного текста будет доступен в памяти, пока сборщик мусора не очистит её. И поскольку String используются в String pool для повторного использования, существует довольно высокая вероятность того, что пароль останется в памяти надолго, что совсем не безопасно. **2. Рекомендации авторов** Java сама по себе рекомендует использовать метод getPassword () из класса JPasswordField, который возвращает char []. **3. Случайная печать в логах** С типом String всегда существует опасность того, что текст, хранящийся в строке будет напечатан в файле логов или в консоли. В то же время в случае использования Array, вы не будете печатать содержимое массива, а только его расположение в памяти.
65	**Почему String неизменяемый и финализированный класс?**	*1. **Для** возможности **реализации строкового пула (String pool)*** *Виртуальная машина имеет возможность сохранить много места в памяти (heap space) т.к. разные строковые переменные указывают на одну переменную в пуле. При изменяемости строк было бы невозможно реализовать интернирование, поскольку если какая-либо переменная изменит значение, это отразится также и на остальных переменных, ссылающихся на эту строку.*

		*2. **Безопасность*** *Изменяемость строк несло бы в себе потенциальную угрозу безопасности приложения. Поскольку в Java строки используются для передачи параметров для авторизации, открытия файлов и т.д. — неизменяемость позволяет избежать проблем с доступом.* *3. **Для многопоточности**. Неизменяемые **строки потокобезопасны*** *Так как строка неизменяемая то, она безопасна для много поточности и один экземпляр строки может быть совместно использован различными потоками. Это позволяет избежать синхронизации для потокобезопасности. Таким образом, строки в Java полностью потокобезопасны.* *4. **Ключ для HashMap*** *Поскольку строка неизменная, её hashcode кэшируется в момент создания и нет никакой необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в Map и его обработка будет быстрее, чем других ключей HashMap. Поэтому строка наиболее часто используется в качестве ключа HashMap.* - можно передавать строку между потоками не опасаясь, что она будет изменена - отсутствуют проблемы с синхронизацией потоков - отсутствие проблем с утечкой памяти - отсутствие проблем с доступом и безопасностью при использовании строк для передачи параметров авторизации, открытия файлов и т.д. - кэширование hashcode - Экономия памяти при использовании пула строк для хранения повторяющихся строк.
66	**Почему строка является популярным ключом в HashMap в Java?**	Поскольку **строки неизменны**, **их хэшкод кэшируется** в момент создания, и **не требует повторного пересчета.**
67	**Что делает метод intern() в классе String?**	**Помещает** строку **в pool строк.**
68	**Можно ли использовать строки в конструкции switch?**	Да, начиная с Java 7 в операторе switch можно использовать строки, ранние версии Java не поддерживают этого. Более подробно: https://javarush.ru/groups/posts/759-java-string-voprosih-k-sobesedovaniju-i-otvetih-na-nikh-ch1 (10) При этом: - участвующие строки чувствительны к регистру; - использование строк в конструкции switch делает код читабельнее, убирая множественные цепи условий if-else - оператор switch использует метод String.equals() для сравнения полученного значения со значениями case, поэтому добавьте проверку на NULL во избежание NullPointerException.
69	**Какая основная разница между String, StringBuffer, StringBuilder?**	String - неизменяемый, потокобезопасный; StringBuffer - изменяемый, потокобезопасный; StringBuilder - изменяемый, потоконебезопасный.
70	**Существуют ли в java многомерные массивы?**	Многомерные массивы в их классическом понимании в java не существуют. Многомерный массив всегда прямоугольный и неразрывен в памяти. А то, что в java считается мнгомерным - в других языках ещё называют "зубчатым массивом" или массивом массивов.

71	**Какими значениями инициируются переменные по умолчанию?**	**byte 0** **short 0** **int 0** **long 0L** **float 0.0f** **double 0.0d** **char '\u0000'** **boolean false** **Обьекты null** Локальные (в методе) переменные не имеют значений по умолчанию, их имеют поля класса. Не static-поле класса будет инициализировано после того, как будет создан объект этого класса. А static-поле будет инициализировано тогда, когда класс будет загружен виртуальной Java машиной.
72	**Что такое сигнатура метода?**	Это **имя метода** плюс **параметры** (порядок параметров имеет значение из-за множественной передачи данных через троеточие, которое должно располагаться последним). В сигнатуру метода не входит возвращаемое значение, а также бросаемые им исключения. А сигнатура метода в сочетании с типом возвращаемого значения и бросаемыми исключениями называется **контрактом метода.**
73	**Расскажите про метод main**	**Является**, как правило, **точкой входа** в программу и **вызывается JVM**. Как только **заканчивается выполнение** метода **main()**, так сразу же **завершается работа** самой **программы**. static - чтобы JVM смогла загрузить его во время компиляции. public static void и сигнатура - обязательное декларирование. Мэйнов может быть много и может не быть вообще. Может быть перегружен.
74	**Каким образом переменные передаются в методы, по значению или по ссылке?**	**Java передает параметры по значению. Всегда.** С примитивами, мы получаем копию содержимого. Со ссылками мы тоже получаем копию ссылки. https://javarush.ru/groups/posts/857-peredacha-parametrov-v-java
	ООП в Java	
76	**Какие виды классов есть в java?**	1. Вложенные классы – нестатические классы внутри внешнего класса. 2. Вложенные статические классы – статические классы внутри внешнего класса. 3. Локальные классы Java – классы внутри методов. разница между локальным и внутреним 4. Анонимные Java классы – классы, которые создаются на ходу. Анонимные классы доступно 5. Final, abstract, enum - классы
77	**Расскажите про вложенные классы. В каких случаях они применяются?**	Нужны для обслуживания внешних классов 1. Статические вложенные классы (Static nested classes) Есть возможность обращения к внутренним статическим полям и методам класса обертки. 2. Вложенные классы Есть возможность обращения к внутренним полям и методам класса обертки. Не может иметь статических объявлений. Внутри такого класса нельзя объявить перечисления. Если нужно явно получить this внешнего класса — OuterClass.this

		3. Локальный класс Видны только в пределах блока, в котором объявлены. Не могут быть объявлены как private/public/protected или static (по этой причине интерфейсы нельзя объявить локально). Не могут иметь внутри себя статических объявлений (полей, методов, классов), но могут иметь константы (static final) Имеют доступ к полям и методам обрамляющего класса. Можно обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final или являются effectively final. 4. Анонимные классы Локальный класс без имени.
78	**- Что такое «локальный класс»? Каковы его особенности?**	Данные классы **объявляются внутри других методов**. Они **обладают всеми свойствами нестатического вложенного класса**, только **создавать их экземпляры можно только в методе**. **Особенности:** Локальные классы **способны работать** только с **final переменными** метода. **С 8+ версий Java можно использовать не final переменные в локальных класса**х, но только при условии, что они **не будут изменяться**. Локальные классы **нельзя объявлять с модификаторами доступа**. Локальные классы **обладают доступом к переменным метода.** Может быть создан внутри блоков инициализации.
79	**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. Анонимные классы всегда являются конечными классами. Каждое объявление анонимного класса уникально. Видны только внутри того метода, в котором определены. **В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового**

		использования».
80	**- Каким образом из вложенного класса получить доступ к полю внешнего класса?**	Статический вложенный класс имеет прямой доступ только к статическим полям обрамляющего класса. Простой вложенный 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 внешнему полю следует через имя внешнего класса. Например: Outer.this.field.
81	**Что такое перечисления (enum)?**	Перечисления представляют **набор логически связанных констант.** Перечисление фактически представляет новый класс, поэтому мы можем определить переменную данного типа и использовать ее. **Перечисления**, как и обычные классы, **могут определять конструкторы, поля и методы.** Следует отметить, что **конструктор по умолчанию приватный.** Также можно определять методы для отдельных констант. Методы: **-ordinal()** возвращает порядковый **номер определенной константы** (нумерация начинается с 0) **-values()** возвращает **массив всех констант** перечисления Enum имеет ряд преимуществ при использовании в сравнении с static final int. Главным отличием является то что **используя enum вы можете проверить тип данных**. **Недостатки** - К ним **не применимы операторы >, <, >=, <=** - enum также **требует больше памяти для хранения** чем обычная константа. Нужны для ограничения области допустимых значений: например, времена года, дни недели
82	**Как проблема ромбовидного наследования решена в java?**	**В Java нет поддержки множественного наследования классов.** Предположим, что SuperClass — это абстрактный класс, описывающий некоторый метод, а классы ClassA и ClassB — обычные классы наследники SuperClass, а класс ClassC наследуется от ClassA и ClassB одновременно. Вызов метода родительского класса приведет к неопределенности, так как компилятор не знает о том, метод какого именно суперкласса должен быть вызван. Это и есть основная причина, почему в Java нет поддержки множественного наследования классов. **1. Классы всегда побеждают**: Определенный в классе / суперклассе метод всегда имеет высший приоритет перед дефолтными методами интерфейсов. **2.** Если не срабатывает правило 1, то **побеждают саб-интерфейсы (more specific)**. Т.е. если интерфейс B наследует A, и у обоих есть методы с одинаковой сигнатурой, то побеждает B. **3.** Если оба правила не работают, то класс, наследующий конфликтующие интерфейсы, **должен явно через super определить**, какой именно метод вызвать, иначе компилятор будет сильно материться.
83	**Что такое конструктор**	Если у какого-либо класса не определить конструктор, то компилятор сгенерирует конструктор без аргументов - так

	по умолчанию?	называемый «конструктор по умолчанию». Если у класса уже определен какой-либо конструктор, то конструктор по умолчанию создан не будет и, если он необходим, его нужно описывать явно.
84	**Могут ли быть приватные конструкторы? Для чего они нужны?**	Да, могут. Приватный конструктор **запрещает создание экземпляра класса вне методов самого класса.** Нужен для реализации паттернов, например singleton.
85	**Расскажите про классы-загрузчики и про динамическую загрузку классов.**	При запуске JVM, используются три загрузчика классов: - **Bootstrap ClassLoader** - главный загрузчик - загружает платформенные классы JDK из архива rt.jar - **AppClassLoader** - системный загрузчик - загружает классы приложения, определенные в CLASSPATH - **Extension ClassLoader** - загрузчик расширений - загружает классы расширений, которые по умолчанию находятся в каталоге jre/lib/ext. **Динамическая загрузка** происходит "на лету" **в ходе выполнения программы** с помощью статического метода **класса Class.forName**(имя класса). Для чего нужна динамическая загрузка? Например мы не знаем какой класс нам понадобится и принимаем решение в ходе выполнения программы передавая имя класса в статический метод forName().
86	**Чем отличаются конструкторы по-умолчанию, конструктор копирования и конструктор с параметрами?**	-У конструктора по умолчанию отсутствуют какие-либо аргументы. -Конструктор копирования принимает в качестве аргумента уже существующий объект класса для последующего создания его клона. -Конструктор с параметрами имеет в своей сигнатуре аргументы (обычно необходимые для инициализации полей класса).
87	**Какие модификаторы доступа есть в Java? Какие применимы к классам?**	**Private** – **доступ** к компоненту только **из этого класса**, в котором объявлен. **Default** – Переменная или метод будут доступны **для любого другого класса в том же пакете.** **Protected** – Поля protected **доступны всем классам внутри пакета,** а также **всем классам-наследникам вне пакета**. **Public** – доступ к компоненту из экземпляра **любого класса и любого пакета.** Класс может быть объявлен с модификатором public и default.
88	**Что означает модификатор static?**	Модификатор static в Java напрямую связан с классом. Если поле статично, значит оно принадлежит классу, если метод статичный — аналогично: он принадлежит классу. Исходя из этого, можно обращаться к статическому методу или полю, используя имя класса. Например, если поле count статично в классе Counter, значит, вы можете обратиться к переменной запросом вида: Counter.count.
89	**Может ли статический метод быть переопределён или перегружен?**	**Нельзя переопределять статические методы.** Если вы объявите такой же метод в классе-наследнике (subclass), т.е. метод с таким же именем и сигнатурой, вы лишь «спрячете» метод суперкласса вместо переопределения. Это явление известно как **сокрытие методов** (hiding methods). **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.

90	**Могут ли нестатические методы перегрузить статические?**	Да. Это будут просто два разных метода для программы. Статический будет доступен по имени класса.
91	**Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?**	При переопределении метода **нельзя сузить модификатор доступа к методу** (например, с public до private), но можно расширить. **Изменить тип возвращаемого значения нельзя**, но **можно сузить** возвращаемое значение, **если они совместимы**. Например, если метод возвращает объект класса, а переопределенный метод возвращает класс-наследник.
92	**Что можно изменить в сигнатуре метода при переопределении? Можно ли менять модификаторы (throws и тп)?**	В сигнатуре(имя + параметры) менять ничего нельзя. **Возможно расширение уровня доступа.** **Изменять тип возвращаемого значения** при переопределении метода разрешено только **в сторону сужения типа** (вместо родительского класса - наследника). **Секцию throws** метода можно не указывать, но стоит помнить, что **она 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.
93	**Могут ли классы быть статическими?**	**Класс можно объявить статическим за исключением классов верхнего уровня.** Такие классы известны как «вложенные статические классы» (nested static class).
94	**Что означает модификатор final? К чему он может быть применим?**	Для класса это означает, что **класс** не сможет **иметь подклассов**, т.е. запрещено наследование. Следует также отметить, что к **abstract-классам нельзя применить модификатор final**, т.к. это взаимоисключающие понятия. Для **переменных примитивного типа** это означает, что **однажды присвоенное значение не может быть изменено** Для **ссылочных переменных** это означает, что после присвоения объекта, нельзя изменить ссылку на данный объект. Важно: **Ссылку изменить нельзя, но состояние объекта изменять можно.** Т.к. массив – это объект, то final означает, что после присвоения ссылки на объект, уже нельзя ее изменить, но можно изменять состояние объекта.
95	**Что такое абстрактные классы? Чем они отличаются от обычных?**	Абстрактным называется класс, на основе которого не могут создаваться объекты. Как обычный класс, но **с абстрактными методами**. **Нельзя создать** объект или **экземпляр абстрактного класса**. Наследниками абстрактного класса могут быть другие абстрактные классы
96	**Может ли быть абстрактный класс без абстрактных методов?**	Класс **может** быть абстрактным без единого абстрактного метода, если у него указан модификатор abstract.
97	**Могут ли быть конструкторы у абстрактных классов? Для чего они нужны?**	**Да. Необходимы для наследников.** В абстрактном классе в Java можно объявить и определить конструкторы. Даже если вы не объявили никакого конструктора, компилятор добавит в абстрактный класс конструктор по умолчанию без аргументов. Абстрактные конструкторы будут часто использоваться для обеспечения ограничений класса или

		инвариантов, таких как минимальные поля, необходимые для настройки класса.
98	**Что такое интерфейсы? Какие модификаторы по умолчанию имеют поля и методы интерфейсов?**	**Интерфейс** — это план класса или, можно сказать, набор абстрактных методов и статических констант. В интерфейсе каждый метод является открытым и абстрактным, но не содержит конструктора. Таким образом, интерфейс в основном представляет собой группу связанных методов с пустыми телами. Другими словами, **интерфейс определяет как элементы будут взаимодействовать между собой**. **- методы интерфейса** являются публичными (**public**) и абстрактными (**abstract**), - **поля — public static final**.
99	**Чем интерфейсы отличаются от абстрактных классов? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?**	**1. Интерфейс описывает** только **поведение** (методы) объекта, а вот **состояний** (полей) у него **нет** (кроме public static final), в то время как у абстрактного класса они могут быть. **2**. Мы **можем наследовать** только **один класс**, **а реализовать интерфейсов — сколько угодно**. Интерфейс может наследовать (extends) другой интерфейс/интерфейсы. 3. **Абстрактные классы используются**, когда **есть** отношение **"is-a"**, то есть класс-наследник расширяет базовый абстрактный класс, а **интерфейсы могут быть реализованы разными классами**, вовсе **не связанными друг с другом**. 4. **Абстрактный класс может реализовывать методы**; **интерфейс** может реализовывать дефолтные методы **начиная с 8й версии**. https://javahelp.online/osnovy/voprosy-otvety-sobesedovanie-java (Q5)
100	**Может ли один интерфейс наследоваться от другого? От двух других?**	**Да**, может. Используется ключевое слово **extends**
101	**Что такое дефолтные методы интерфейсов? Для чего они нужны?**	В **JDK 8** была добавлена такая функциональность как **методы по умолчанию с модификатором default.** И теперь интерфейсы могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Это **нужно для обратной совместимости**. (Если один или несколько методов добавляются к интерфейсу, все реализации также будут вынуждены их реализовывать. Методы интерфейса по умолчанию являются эффективным способом решения этой проблемы.)
102	**Как решается проблема ромбовидного наследования при наследовании интерфейсов при наличии default методов?**	класс, наследующий конфликтующие интерфейсы, **должен явно через super определить**, какой именно метод вызвать: **InterfaceB.super.method();**
103	**Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?**	**1. Статические блоки** от первого до последнего предка(от предка до наследника) **2. Попарно динамической блок инициализации и конструктор** от первого до последнего предка
10	**Зачем нужны и какие**	Инициализация - это когда мы впервые задаем переменной какое-либо значение.

4	**бывают блоки инициализации?**	Существуют **статические** и **нестатические** блоки инициализации.
105	**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 принадлежит только самому классу.
106	**Что произойдет, если в блоке инициализации возникнет исключительная ситуация?**	**Для нестатических** блоков инициализации, **если выбрасывание исключения прописано явным образом, требуется**, чтобы **объявления** этих исключений были перечислены **в throws всех конструкторов класса**. **Иначе** будет **ошибка компиляции**. **Для статического** блока выбрасывание исключения в явном виде, приводит к **ошибке компиляции**.
107	**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**. **Если** исключение: java.lang.**ThreadDeath** - смерть потока. В этом случае никакое **исключение** выброшено **не будет**.
108	**Что такое класс Object?**	**Базовый класс для всех остальных объектов в Java. Любой класс наследуется от Object и, соответственно, наследуют его методы** **Все классы являются наследниками** суперкласса **Object**. Это не нужно указывать явно. В результате объект **Object может ссылаться на объект любого** другого **класса**. **Рефлексия** (от позднелат. reflexio - обращение назад) - это **механизм исследования данных** о программе **во время её выполнения.**
109	**Какие методы есть у класса Object (перечислить все)? Что они делают?**	**- equals()** - проверка на равенство двух обьектов **- hashCode()** - изначально случайно число int **- toString()** - представления данного объекта в виде строки. **- getClass()** - получение типа данного обьекта **- clone() -** клонирует объект методом. **- finalize() -** deprecated, вызывается GC перед удалением. (нет гарантии что будет вызван) **для многопоточки** **wait():** освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify() **notify():** продолжает работу потока, у которого ранее был вызван метод wait() **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван метод wait() **- wait(long timeOut)** - нить освобождает монитор и «становится на паузу»,принимает максимальное время ожидания в миллисекундах. **- wait(long timeOut, int nanos)** - нить освобождает монитор и «становится на паузу»,принимает максимальное время ожидания в

		миллисекундах, дополнительное время, в диапазоне наносекунд 0-999999.
110	**Расскажите про equals и hashcode**	**Хеш-код** — это **целочисленный результат** работы **метода**, **которому** в качестве входного параметра **передан объект.** **Если более точно, то это битовая строка фиксированной длины, полученная из массива произвольной длины.** **Equals** - это метод, определенный в Object, который служит для сравнения объектов. **При сравнении объектов при помощи ==** идет **сравнение по ссылкам**. **При сравнении по equals**() идет **сравнение по состояниям** объектов. **Свойства equals**(): • **Симметричность**: Для двух ссылок, a и b, a.equals(b) тогда и только тогда, когда b.equals(a) • **Рефлексивность**: для любого заданного значения x, выражение x.equals(x) должно возвращать true. Заданного — имеется в виду такого, что x != null • **Постоянство**: повторный вызов метода equals() должен возвращать одно и тоже значение до тех пор, пока какое-либо значение свойств объекта не будет изменено. • **Транзитивность**: Если a.equals(b) и b.equals(c), то тогда a.equals(c) • **Совместимость с hashCode**(): Два тождественно равных объекта должны иметь одно и то же значение hashCode() При переопределении equals() обязательно нужно переопределить метод hashCode(). Равные объекты должны возвращать одинаковые хэш коды.

111	**Каким образом реализованы методы hashCode() и equals() в классе Object?**	1 - Реализация метода Object.equals() сводится к проверке на равенство двух ссылок:
		
		public boolean equals(Object obj) {
		return (this == obj);
		}
		
		2 - HashCode реализован таким образом, что для одного и того же входного объекта, хеш-код всегда будет одинаковым.
		Реализация метода Object.hashCode() описана как native, т.е. написана не на Java. Непереопределенный hashCode возвращает идентификационный хеш, основанный на состоянии потока, объединённого с xorshift (в OpenJDK8). А вообще, функция предлагает шесть методов на базе значения переменной hashCode.
		
		0. Случайно сгенерированное число.
		1. Функция адреса объекта в памяти.
		2. Жёстко запрограммированное значение 1 (используется при тестировании на чувствительность (sensitivity testing)).
		3. Последовательность.
		4. Адрес объекта в памяти, приведённый к целочисленному значению.
		5. Состояние потока, объединённое с xorshift (https://en.wikipedia.org/wiki/Xorshift)
		
		public native int hashCode();
		
		Ситуация, когда у разных объектов одинаковые хеш-коды

		называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.
112	**Зачем нужен equals(). Чем он отличается от операции ==?**	**equals**() - сравнение **по состоянию**, **==** - **по ссылкам**
113	**Правила переопределения equals()**	1. Проверить на равенство ссылки объектов this и параметра метода o. if (this == o) return true; 2. Проверить, определена ли ссылка o, т. е. является ли она null. Если в дальнейшем при сравнении типов объектов будет использоваться оператор instanceof, этот пункт можно пропустить, т. к. этот параметр возвращает false в данном случае null instanceof Object. 3. Сравнить типы объектов this и o с помощью оператора instanceof или метода getClass(), руководствуясь описанием выше и собственным чутьем. 4. Если метод equals переопределяется в подклассе, не забудьте сделать вызов super.equals(o) 5. Выполнить преобразование типа параметра o к требуемому классу. 6. Выполнить сравнение всех значимых полей объектов: o для примитивных типов (кроме float и double), используя оператор == o для ссылочных полей необходимо вызвать их метод equals o для массивов можно воспользоваться перебором по циклу, либо методом Arrays.equals() o для типов float и double необходимо использовать методы сравнения соответствующих оберточных классов Float.compare() и Double.compare()
114	**что будет если переопределить equals() и не переопределить hashcode()**	Нарушится контракт. Классы и методы, которые использовали правила этого контракта могут некорректно работать. Так для объекта HashMap это может привести к тому, что пара, которая была помещена в Map возможно не будет найдена в ней при обращении к Map, если используется новый экземпляр ключа.
115	**Какой контракт между hashCode() и equals()?**	1) Если два объекта возвращают разные значения hashcode(), то они не могут быть равны 2) Если equals объектов true, то и хэшкоды должны быть равны. 3) Переопределив equals, всегда переопределять и hashcode.
116	**Для чего нужен метод hashCode()?**	вычисляет целочисленное значение для конкретного элемента класса, чтобы использовать его для быстрого поиска и доступа к этому элементу в hash-структурах данных, например, HashMap, HashSet и прочих.
117	**- Правила переопределения метода hashcode().**	Если хеш-коды разные, то и входные объекты гарантированно разные. Если хеш-коды равны, то входные объекты не всегда равны. При вычислении хэш-кода следует использовать те же поля, которые сравниваются в equals и которые не вычисляются на основе других значений. - вызов метода hashCode один и более раз над одним и тем же объектом должен возвращать одно и то же хэш-значение, при условии что поля объекта, участвующие в вычислении значения, не изменялись. - вызов метода hashCode над двумя объектами должен всегда возвращать одно и то же число, если эти объекты равны (вызов метода equals для этих объектов возвращает true).

		- вызов метода hashCode над двумя неравными между собой объектами должен возвращать разные хэш-значения. Хотя это требование и не является обязательным, следует учитывать, что его выполнение положительно повлияет на производительность работы хэш-таблиц.
118	**- Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode()?**	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().
119	**Могут ли у разных объектов быть одинаковые hashCode()?**	Когда у разных объектов одинаковые хеш-коды называется — **коллизией**.
120	**Почему нельзя реализовать hashcode() который будет гарантированно уникальным для каждого объекта?**	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
121	**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, а это дает гораздо лучшую хэш-функцию.
122	**Чем a.getClass().equals(A.class) отличается от a instanceOf A.class**	getClass() получает только класс, а оператор instanceof проверяет является ли объект экземпляром класса или его потомком
	Исключения	
124	**Что такое исключения?**	Исключение — **это ошибка (является объектом),** возникающая во время выполнения программы.
125	**Опишите иерархию исключений.**	1. класс **Throwable** (checked) 2. от **Throwable -> Error** (**ошибки JVM**) и **Exception** (**checked** общие) 3. от **Exception** **- > RuntimeException (unchecked)** **- > IOException, SQLException, ReflectiveOperationException (checked)** **4.RuntimeException (unchecked):** ClassCastExceptiuon IndexOutOfBoundException AritthmeticException NullPointerException checked - зависит от программиста, unchecked - от программиста не

		зависит
126	**Расскажите про обрабатываемые и необрабатываемые исключения**	1. Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными. 2. Unchecked исключения в Java — наследованные от RuntimeException, checked — от Exception. Checked исключения отличаются от Unchecked исключения в Java, тем что наличие\обработка Checked исключения проверяются компилятором на этапе компиляции. Наличие\обработка Unchecked исключения происходит на этапе выполнения.
127	**Можно ли обработать необрабатываемые исключения?**	Можно, чтобы в некотрых случаях программа не прекратила работу
128	**Какой оператор позволяет принудительно выбросить исключение?**	Throw
129	**О чем говорит ключевое слово throws?**	Метод потенциально может выбросить исключение с указанным типом. Передаёт обработку исключения вышестоящему методу.
130	**Как создать собственное («пользовательское») исключение?**	Необходимо унаследоваться от базового класса требуемого типа исключений (например, от Exception или RuntimeException). и переопределит методы
131	**Расскажите про механизм обработки исключений в java (Try-catch-finally)**	**Try - блок в котором может появиться исключение;** **Catch - блок в котором мы указываем исключение и логику его обработки;** **Finally - блок который обязательно отработает**
132	**Возможно ли использование блока try-finally (без catch)?**	try может быть в паре с finally, без catch. Работает это точно так же - после выхода из блока try выполняется блок finally
133	**Может ли один блок catch отлавливать сразу несколько исключений?**	**Да**
134	**Всегда ли выполняется блок finally? Существуют ли ситуации, когда блок finally не будет выполнен?**	Да, кроме случаев завершения работы программы или JVM: 1 - Finally может не выполниться в случае если в блоке try вызывает System.exit(0), 2 - Runtime.getRuntime().exit(0), Runtime.getRuntime().halt(0) и если во время исполнения блока try виртуальная машина выполнила недопустимую операцию и будет закрыта. 3 - В блоке try{} бесконечный цикл.
135	**Может ли метод main() выбросить исключение во**	Может и оно будет передано в виртуальную машину Java (JVM). Для случая с методом main произойдет две вещи: - будет завершен главный поток приложения;

	вне и если да, то где будет происходить обработка данного исключения?	- будет вызван ThreadGroup.uncaughtException.
136	**В каком порядке следует обрабатывать исключения в catch блоках?**	**От наследника к предку**
137	**Что такое механизм try-with-resources?**	Дает **возможность объявлять** один или **несколько ресурсов в блоке try, которые будут закрыты автоматически** без использования finally блока. В **качестве ресурса** можно использовать любой **объект**, класс **которого реализует** интерфейс java.lang.**AutoCloseable или** java.io.**Closeable**.
138	**Что произойдет если исключение будет выброшено из блока catch после чего другое исключение будет выброшено из блока finally?**	**finally**-секция **может** «**перебить**» **throw/return при помощи** другого **throw/return**
139	**Что произойдет если исключение будет выброшено из блока catch после чего другое исключение будет выброшено из метода close() при использовании try-with-resources?**	В **try-with-resources** добавленна возможность хранения "подавленных" исключений, и брошенное try-блоком исключение **имеет больший приоритет**, **чем исключения** получившиеся **во время закрытия**.
	Сериализация и копирование	
141	**Что такое сериализация и как она реализована в Java?**	Сериализация это процесс сохранения состояния объекта в последовательность байт; Реализована через интерфейс - маркер Serializable.
142	**Для чего нужна сериализация?**	Для компактного сохранения состояния объекта и считывание этого состояния.
143	**Опишите процесс сериализации/десериализации с использованием Serializable.**	1) Класс объекта должен реализовывать интерфейс Serializable 2) Создать поток ObjectOutputStream (oos), который записывает объект в переданный OutputStream. 3) Записать в поток: oos.writeObject(Object); 4) Сделать oos.flush() и oos.close()

3		
144	**Как изменить стандартное поведение сериализации/десериализации?**	Использовать интерфейс **Externalizable.** Переопределить методы writeExternal(ObjectOutput out) throws IOException readExternal(ObjectInput in) throws IOException, ClassNotFoundException
145	**Какие поля не будут сериализованы при сериализации? Будет ли сериализовано final поле?**	1) Добавить к полю модификатор **transient**. В таком случае после восстановления его значение будет null. 2) Сделать поле **static**. Значения статических полей автоматически не сохраняются. 3) Поля с модификатором **final** сериализуются **как** и **обычные**. **За** одним **исключением** – их **невозможно десериализовать при** использовании **Externalizable**, поскольку final-поля должны быть инициализированы в конструкторе, а после этого в readExternal изменить значение этого поля будет невозможно. Соответственно, если необходимо сериализовать объект с final-полем неоходимо использовать только стандартную сериализацию.
146	**Как создать собственный протокол сериализации?**	Для создания собственного протокола нужно просто **переопределить writeExternal**() и **readExternal**(). В отличие от двух других вариантов сериализации, здесь ничего не делается автоматически. Протокол полностью в ваших руках.
147	**Какая роль поля serialVersionUID в сериализации?**	Поле private static final long serialVersionUID содержит **уникальный идентификатор** версии сериализованного класса. Оно вычисляется по содержимому класса - полям, их порядку объявления, методам, их порядку объявления. Соответственно, **при любом изменении** в классе **это поле поменяет** свое **значение**. Если мы не объявляем его явно, Java делает это за нас.
148	**Когда стоит изменять значение поля serialVersionUID?**	Вы должны изменить serialVersionUID только тогда, **когда** вы **сознательно хотите нарушить совместимость со** всеми **существующими сериализациями** , например, когда изменения в вашем классе сделают его настолько семантически отличным, что у вас не будет выбора - в этом случае вы действительно должны несколько раз подумать о том, что вы на самом деле делаете.
149	**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 не совсем нужно.

		- Решение - **В классе определяется метод с сигнатурой "Object readResolve() throws ObjectStreamException"** **- Назначение -** этого метода - **возвращать замещающий объект вместо объекта**, **на котором** он **вызван**.
150	**Расскажите про клонирование объектов.**	в Java, есть 3 способа клонирования объекта: 1. **С использованием** интерфейса **Cloneable**; Первый способ подразумевает, что вы будете использовать механизм так называемого «поверхностного клонирования» и сами позаботитесь о клонировании полей-объектов. Метод clone() в родительском классе Object является protected, поэтому требуется переопределение его с объявлением как public. Он возвращает экземпляр объекта с копированными полями-примитивами и ссылками. И получается что у оригинала и его клона поля-ссылки указывают на одни и те же объекты. 2. **С использованием конструктора** клонирования **объекта**; В классе описывается конструктор, который принимает объект этого же класса и инициализирует значениями его полей поля нового объекта. 3. **С использованием сериализации.** Он заключается в сохранении объекта в поток байтов с последующей эксгумацией его от туда.
151	**В чем отличие между поверхностным и глубоким клонированием?**	**Поверхностное копирование копирует настолько малую часть информации, насколько это возможно. По умолчанию**, **клонирование** в Java **является поверхностным**, т.е. Object class не знает о структуре класса, которого он копирует. **Глубокое копирование дублирует все.** Глубокое копирование — это **две коллекции**, в **одну** из которых **дублируются все элементы оригинальной коллекции.**
152	**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.
153	**Почему метод clone() объявлен в классе Object, а не в интерфейсе Cloneable?**	Метод clone() объявлен в классе Object с сигнатурой native, чтобы обеспечить доступ к стандартному механизму "поверхностного копирования" объектов (копируются значения всех полей, включая ссылки на сторонние объекты); он объявлен, как protected, чтобы нельзя было вызвать этот метод у не переопределивших его объектов.
154	**Как создать глубокую копию объекта? (2 способа)**	Глубокое клонирование требует выполнения следующих правил: -Нет необходимости копировать отдельно примитивные данные; -Все классы-члены в оригинальном классе должны поддерживать клонирование. Для каждого члена класса должен вызываться super.clone() при переопределении метода clone(); -Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному. **1 Сериализация** – это еще один способ глубокого копирования. Мы

	просто сериализуем нужный объект и десериализуем его. Очевидно, объект должен поддерживать интерфейс Serializable. Мы сохраняет объект в массив байт и потом прочитать из него. **2 При помощи библиотеки DeepCloneable** Глубокое клонирование с этой библиотекой сводится с двум строкам кода: Cloner cloner = new Cloner(); DeepCloneable clone = cloner.deepClone(this);