**что такое ООП.**

Объе́ктно-ориенти́рованное программи́рование — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Плюсы

* Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
* Данные и операции вместе образуют определенную сущность и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
* Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
* Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

**принципы ООП**

**Полиморфизм** — это способность объекта принимать различные формы. Наиболее распространенное использование полиморфизма в ООП происходит, когда ссылка на родительский класс используется для ссылки на объект дочернего класса.

Полиморфизм - это возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

**Наследование** - это механизм, позволяющий создавать классы на  основе другого класса.

**Инкапсуляция**

**https://javarush.ru/groups/posts/1969-principih-inkapsuljacii**

Абстракция – веделение общих параметров у сущностей.

**Модификаторы доступа**

* public - "публичный, доступный всем"
* default - "по умолчанию". Когда мы не пишем модификатора доступа (как мы это делали в наших предыдущих уроках), он по умолчанию имеет значение default. Данные с этим модификатором видны в пределах package.
* protected - "защищенный". На самом деле это то же самое, что и default, только доступ имеют еще и классы-наследники даже если они в других пакетах
* private - "частный, личный". Такие данные видны только самому классу.

**4) Указатели this и super.**

Так вот, как я уже сказал в начале, главное отличие между this и super в Java в том, что this представляет текущий экземпляр класса, в то время как super - текущий экземпляр родительского класса. Вот один из примеров использования переменных this и super — вы наверняка уже видели примеры вызовов конструкторов одного из другого, т.н. вызовы конструкторов по цепочке, это возможно благодаря использованию ключевых слов this и super. Внутри класса для вызова своего конструктора без аргументов используется this(), тогда как super()используется для вызова конструктора без аргументов, или как его ещё называют, конструктора по умолчанию родительского класса. Между прочим, таким способом вызывать можно не только конструктор без аргументов, а и вообще любой другой конструктор, передав ему соответствующие параметры.

Ещё this и super в Java используются для обращения к переменным экземпляра класса и его родителя. Вообще-то, к ним можно обращаться и без префиксов super и this, но только если в текущем блоке такие переменные не перекрываются другими переменными, т.е. если в нем нет локальных переменных с такими же именами, в противном же случае использовать имена с префиксами придется обязательно, но это не беда, т.к. в таком виде они даже более читабельны. Классическим примером такого подхода является использование this внутри конструктора, который принимает параметр с таким же именем, как и у переменной экземпляра. Дальше в статье мы узнаем, какие ещё есть отличия между super и this, и рассмотрим некоторые примеры их использования.

**Чем this и super похожи**

Прежде чем рассматривать отличия ключевых слов this и super, давайте посмотрим на некоторые их сходства:

* И this, и super — это нестатические переменные, соответственно их нельзя использовать в статическом контексте, а это означает, что их нельзя использовать в методе main. Это приведет к ошибке во время компиляции "на нестатическую переменную this нельзя ссылаться из статического контекста". То же самое произойдет, если в методе main воспользоваться ключевым словом super.
* И this, и super могут использоваться внутри конструкторов для вызова других конструкторов по цепочке, нпр., this() и super() вызывают конструктор без аргументов наследующего и родительского классов соответственно.

**5) Модификатор static**

Иногда требуется определить член класса, который будет использоваться независимо от любого объекта этого класса. И его можно использовать самостоятельно без ссылки на конкретный экземпляр. Для создания подобного члена класса, нужно в начало его объявления перед объявлением типа поместить ключевое слово static. В этом случае он становится доступен до создания каких-либо объектов его класса и без ссылки на какой-либо объект. Статическими могут быть и переменные и методы.

По существу переменные экземпляров, объявленные как static, являются глобальными переменными. При объявлении объектов их класса программа не создаёт никаких копий статической переменной. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

У статических методов есть ряд ограничений:

они могут вызывать только другие статические методы

они могут непосредственно осуществлять доступ только к статическим переменным

они не могут ссылаться на члены типа this или super

**6) Наследование и реализация.**

Наследование (англ. inheritance) - это механизм, позволяющий создавать классы на  основе другого класса.

Так, один класс может "наследовать" характеристики другого - его методы и переменные.

* Класс, являющийся основой, называют: базовым, супер, родительским.
* Класс, который создают, называют: потомок, наследник или производный класс.

***Правила наследования***

**Правило 1. Наследуем только один класс.**

Java не поддерживает наследование нескольких классов. Один класс - один родитель.

Обратите внимание - нельзя наследовать самого себя!

**Правило 2. Наследуется все кроме приватных переменных и методов.**

Выше мы говорили, что класс-наследник будет иметь доступ ко всем переменным и методам родителя. Это не совсем так.

На самом деле, все методы и переменные, помеченные модификатором private, не доступны классу-наследнику.

**Правило 3. Переделать метод класса-родителя.**

Представим, что мы наследуем класс, но нам нравится не все, что мы унаследовали. Допустим мы хотим, чтобы определенный--- метод работал не так, как в родителе.

**Правило 4. Вызываем методы родителя через ключевое слово super.**

Представим, что Вы хотите изменить метод родительского класса совсем чуть-чуть - буквально дописать пару строк. Тогда в своем методе мы можем вызвать родительский метод с помощью ключевого слова super.

**Правило 5. Запрещаем наследование.**

Если Вы не хотите, чтобы кто-то наследовал Ваш класс, поставьте перед ним модификатор final.

**7) Интерфейсы.**

В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово **interface.**

 Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**

При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

В тоже время мы не можем напрямую создавать объекты интерфейсов

Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом **default**.

Начиная с JDK 8 в интерфейсах доступны статические методы - они аналогичны методам класса:

Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод

По умолчанию все методы в интерфейсе фактически имеют модификатор public. Однако начиная с Java 9 мы также можем определять в интерфейсе методы с модификатором private. Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию.

Подобные методы могут использоваться только внутри самого интерфейса, в котором они определены. То есть к примеру нам надо выполнять в интерфейсе некоторые повторяющиеся действия, и в этом случае такие действия можно выделить в приватные методы

**8) Анонимные классы.**

внутри Java-машины происходит следующее:

1. Создается безымянный Java-класс, реализующий интерфейс MonitoringSystem.
2. Компилятор, увидев такой класс, требует от тебя реализовать все методы интерфейса MonitoringSystem (мы это и сделали 3 раза).
3. Создается один объект этого класса.

«Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».

Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным:

Анонимные классы могут быть созданы:

1) как реализация интерфейса

2) как наследник определённого класса

https://javarush.ru/groups/posts/2193-anonimnihe-klassih

**9) Класс Object и класс Class.**

В Java есть специальный суперкласс **Object** и все классы являются его подклассами. Поэтому ссылочная переменная класса **Object** может ссылаться на объект любого другого класса. Так как массивы являются тоже классами, то переменная класса **Object** может ссылаться и на любой массив.

equals hashCode getclass clone finalize tostring

В Java почти все сущности являются объектами, за исключением примитивных типов. У каждого объекта есть класс. Сами классы тоже является объектами, и они принадлежат классу Class.

У класса Class нет публичных конструкторов. Class - это generic тип. Методы Class предназначены для получения информации о классе (объекте типа Class). Например, можно узнать полное имя класса, какие у него аннотации, какие конструкторы и т.п. Эти методы нужны для reflection. С помощью reflection вы можете создавать объекты, которые принадлежат этому классу, и при этом заранее класс объекта вы можете не знать.

Существуют библиотеки, которые позволяют создавать объекты типа Class "на лету", т.е. вы можете создать новый класс прямо во время работы программы и так же можете изменить существующий класс.

**10) Контракт между  equals() и hashCode().**

**Контракт equals**

При переопределении метода equals разработчик должен придерживаться основных правил, определенных в спецификации языка Java.

Рефлексивность

для любого заданного значения x, выражение x.equals(x) должно возвращать true.  
Заданного — имеется в виду такого, что x != null

Симметричность

для любых заданных значений x и y, x.equals(y) должно возвращать true только в том случае, когда y.equals(x) возвращает true.

Транзитивность

для любых заданных значений x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, x.equals(z) должно вернуть значение true.

Согласованность

для любых заданных значений x и y повторный вызов x.equals(y) будет возвращать значение предыдущего вызова этого метода при условии, что поля, используемые для сравнения этих двух объектов, не изменялись между вызовами.

Сравнение null

для любого заданного значения x вызов x.equals(null) должен возвращать false.

Правила переопределения hashCode

Хэш — это некоторое число, генерируемое на основе объекта и описывающее его состояние в какой-то момент времени. Это число используется в Java преимущественно в хэш-таблицах, таких как HashMap. При этом хэш-функция получения числа на основе объекта должна быть реализована таким образом, чтобы обеспечить относительно равномерное распределение элементов по хэш-таблице. А также минимизировать вероятность появления коллизий, когда по разным ключам функция вернет одинаковое значение.

Исходя из описанных выше контрактов следует, что переопределяя в своем коде метод equals, необходимо всегда переопределять и метод hashCode. Так как фактически два экземпляра класса отличаются, потому что находятся в разных областях памяти, сравнивать их приходится по некоторым логическим признакам. Соответственно, два логически эквивалентных объекта, должны возвращать одинаковое значение хэш-функции.

**Контракт hashCode**

Для реализации хэш-функции в спецификации языка определены следующие правила:

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.

**https://javarush.ru/groups/posts/1989-kontraktih-equals-i-hashcode-ili-kak-ono-vsje-tam**

1. **Что такое полиморфизм? Приведите пример полиморфизма?**

Полиморфизм - это возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

Обратите внимание интервьюера, что при работе с программным кодом мы можем использовать не только объекты типа, который мы назначили при объявлении, но и его наследников. *Это важный момент*: мы можем работать со многими типами, как с одним (при условии, что эти типы являются производными от базового типа). Также это значит, что, объявив переменную типа суперкласса, мы можем присвоить ей значение одного из наследников.

Что нам даёт полиморфизм в Java?

Вопрос, что дает нам использование полиморфизма скорей всего тоже будет. Тут можно отвечать кратко, особо не лазя в дебри:

1. Позволяет подменять реализации объектов. На этом основано тестирование.
2. Обеспечивает расширяемость программы — становится гораздо легче создавать задел на будущее. Добавление новых типов на основе существующих — наиболее частый способ расширения функциональности программ, написанных в ООП стиле.
3. Позволяет объединять объекты с общим типом или поведением в одну коллекцию или массив и управлять ими единообразно (как в наших примерах, заставляя всех танцевать – метод dance или плыть – метод swim).
4. Гибкость при создании новых типов: вы можете выбирать реализацию метода из родителя или переопределить его в потомке.

<https://javarush.ru/groups/posts/polimorfizm-v-java>

1. **Как реализована неизменность String?**
2. **Как реализовать свой Immutable тип данных?**

Не предоставляйте «setter» - методы, которые изменяют поля или объекты, на которые ссылаются поля.

Сделайте все поля final и private.

Не позволяйте подклассам переопределять методы. Самый простой способ сделать это - объявить класс final. Более сложный подход - сделать конструктор закрытым и создавать экземпляры в фабричных методах.

Если поля экземпляра содержат ссылки на изменяемые объекты, не позволяйте изменять эти объекты:

Не предоставляйте методы, которые изменяют изменяемые объекты.

Не делитесь ссылками на изменяемые объекты. Никогда не храните ссылки на внешние изменяемые объекты, переданные конструктору; при необходимости создайте копии и сохраните ссылки на копии. Точно так же создавайте копии ваших внутренних изменяемых объектов, когда это необходимо, чтобы избежать возврата оригиналов в ваших методах.

1. **В чём разница между instanceOf и getClass?**

Причина, по которой производительность instanceof и getClass() == ... различна, заключается в том, что они делают разные вещи.

* instanceof проверяет, является ли ссылка на объект с левой стороны (LHS) экземпляром типа с правой стороны (RHS) или каким-либо подтипом .
* getClass() == ... проверяет, идентичны ли типы.

1. **Могут ли нестатические методы перегрузить статические?**
2. **что такое сигнатура метода?**

Сигнатура метода — это имя метода плюс параметры (причем порядок параметров имеет значение). В сигнатуру метода не входит возвращаемое значение, а также бросаемые им исключения.

1. **private методы в интерфейсах**

С приходом Java 9 стало возможным создание приватных методов в интерфейсах, делая их логическим продолжением default и static методов. Зачем нужны еще и приватные методы? К примеру, поддержание clean code посредством вынесения общей логики default и static методов в приватные.

Приватные методы могут быть либо не статическими либо статическими, так же они обязаны иметь реализацию.

1. **Раннее и позднее связывание**

Чтобы выяснить, в чем состоит различие между ранним (статическим) и поздним (динамическим) связыванием в Java, нужно сначала понять, что такое это самое связывание. Связывание означает наличие связи между ссылкой и кодом.

фундаментальное различие между раним и поздним связыванием в Java состоит в том, что первое происходит рано, во время компиляции на основе типа ссылочной переменной, а второе – позднее, во время выполнения, с использованием конкретных объектов.

**Разница между инкапсуляцией и сокрытием**

<https://ru.gadget-info.com/difference-between-data-hiding>

**перегрузка метода в Java?**

Перегрузка методов — это возможность создавать несколько методов с одинаковым названием, но разными параметрами.

Перегрузка методов - это часть такой составляющей ООП, как полиморфизм.

Перегрузка методов выполняет несколько функций:

1. Вы можете создать несколько конструкторов - с одним, двумя параметрами, и вообще без параметров. Программа будет выбирать тот, который подходит под заданные параметры.
2. В первом примере мы говорили о конструкторах. Но то же самое применимо и к обычным методам. Одним из главных плюсов является то, что Вам не нужно запоминать названия кучи методов, выполняющих похожую функцию.  Представим, что мы хотим создать метод "add", общий для всех типов. У нас есть int, double и String:

Как Вы видите, благодаря перегрузке методов мы можем запомнить один метод - add - и с помощью него выполнить 3 операции. Будь это другой язык программирования, который не поддерживает перегрузку, необходимо было бы создать (и запомнить) методы addInt, addDouble и, например, addStr. И так с каждой операцией!

**что можно менять при перегрузка?**

Можно менять все, кроме имени.

**можно ли менять возвращаемый тип значения?**

да

**перегрузка применима только к методам?**

к конструкторам тоже

**что такое переопределение метода?**

Переопределение метода (англ. Method overriding) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

**что можно делать а что нет при переопределении метода?**

У переопределенного метода должны быть те же аргументы, что и у метода родителя.

1. Если метод voice родительского класса принимает на вход String, переопределенный метод в классе-потомке тоже должен принимать на вход String, иначе компилятор выдаст ошибку:
2. У переопределенного метода должны быть тот же тип возвращаемого значения, что и у метода родителя. В ином случае мы получим ошибку компиляции:

можем только сужать тип параметров (нельзя наследнику кинуть предка)

**что такое Object?**

базовый класс java от которого образуются другие классы

**методы класса object**

* equals
* hashCode
* getclass
* clone
* finalize
* tostring

**для чего нужен finalize**

для сбора мусора,

**какие рекомендации для использования метода finalize**

Java-машина сама решает – вызвать данный метод или нет. Чаще всего объекты, созданные в методе и объявленные мусором после его завершения, уничтожаются сразу же и без всяких вызовов метода finalize(). Этот метод скорее дополнительная страховка, чем надёжное решение. Лучшим вариантом будет освобождать любые используемые ресурсы (обнулять сохраненные ссылки на другие объекты), когда наш объект ещё жив. О пользе этого метода и нюансах его работы я расскажу тебе уровней через 10, или 20. А пока что ты должен знать две вещи: такой метод существует, и (сюрприз!) не всегда вызывается.

**что такое static и что может быть static**

классы(только не верхнего уровня), методы, переменные.

**почему статиком не может быть класс верхнего уровня?**

**можно ли вызывать в статическом методе нестатические?**

нет static во время компиляции, обычные в runtime

**как выйти из ситуации когда из статическом методе необходим доступ до нестатического метода**

Создать экземпляр класса и вызвать метод

**что такое final**

В языке программирования Java ключевое слово final используется в нескольких контекстах для определения объекта, который может быть назначен только один раз

**если final стоит на поле?** Нельзя изменять его значения

**если у нас поле final это массив ?** final не применяется к массивам. Так как final применяется не к самим элементам а к ссылке.

**final на методе что это значит?**

Нельзя переопределить

**если final на классе?**

Нельзя наследоваться от него

**можно ли не в статическом методе вызвать статический?**

можно

**что такое констуктор**

Конструкторы - это специальные методы, которые вызывается при создании объекта. Они "конструируют" новый объект определенного класса.

**нужно ли создавать конструктор если мы не хотим передавать туда никаких значений**

Нет, конструктор всегда существует по умолчанию

**а если напишем свой конструктор который принимает параметры, будет ли констуктор по умолчанию существовать**

Не будет. Нам нужно будет создать заново конструктор по умолчанию, если он нам нужен.

**можно ли в классе вызвать один конструктор из другого?**

да через this

**как вызвать конструктор родителя?**

С помощью ключевого слова super

**что такое пакет?**

папка которая объединяет с логической точки зрения все классы

**какую проблему решают пакеты?**

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

**что такое класс?**

Шаблон для нашего объекта

**что такое объект ?**

это экземпляр класса с определенным состоянием, характеристиками и поведением.

**что такое коллизия для hashCode**

Ситуация, когда у разных объектов одинаковые хеш-коды называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

**Когда объекты разные но показывает одинаковый hashcode**.

У integer диапазон от -2147483648 до 2147483647, т.е. округлив получаем 4 миллиарда разных целых чисел.

А теперь представим ситуацию, у вас 8-10 миллиардов объектов. Вопрос: как каждому из них дать уникальный хеш код используя диапазон в 4 миллиарда?

**если неверно написано вычисление hashCode**

**когда сравниваем объекты через двойное равно что мы сравниваем?**

Ссылки на объект

**что означает анотация Override?**

Это аннотация пишется над методами, которые переопределяются из супер класса.  
  
Это что-то вроде страховки, гарантирующая что 100% переопределен метод.  
  
Компилятор, когда встретит данную аннотацию, проверит, чтоб сигнатуры методов в супер классе и в под классе совпадали, т.е. вы переопределили метод, и в полиморфизме будит вызываться переопределенный метод.  
  
Если вдруг вы ошиблись в сигнатуре метода (например пропустили букву в название), то компилятор выдаст ошибку

**в какой области памяти хранятся примитивы?** Стек. ссылки на объекты тоже

**в какой области хранятся объекты?** - куча

**что такое var args?** - параметр произвольной длины, устанавливается в конце перечисления всех параметров

**что такое абстрактный метод?**

Такие методы определяются с помощью ключевого слова abstract и не имеют никакой реализаци

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

**абстрактный метод может быть только в абстрактном классе ?** еще и в интерфейсе

**абстрактный класс?**

Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

**констуктор абстрактный класс может иметь?** да но только в наследнике

**может абстрактный класс может быть без абстрактных методов?** может

**зачем нужен интерфейс если есть абстрактный класс?** у интефейсов нет состояний

**когда лучше использовать интерфейс а когда абстрактный класс?**

если тесная связь то абстрактный класс, а если нет особой связи то интерфейс

**назови 5 отличий интефейса от абстрактного класса?**

1. Интерфейс описывает только поведение. У него нет состояния. А у абстрактного класса состояние есть: он описывает и то, и другое.
2. Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.
3. Классы могут реализовывать сколько угодно интерфейсов, но наследоваться можно только от одного класса.
4. У интерфейса нет полей(но с 8 явы есть public static final)
5. в абстрактном классе может быть описано тело метода - в интерфейсе нет
6. в абстрактных классах могут быть конструкторы
7. у интефейса только Public модификатор методов

**можно ли создавать static методы в интерфейсе?**

можно но только должны быть сразу реализованы

**могут ли быть поля в интерфейсе?**

да только public static final

**что такое нативные методы?**

Нативные методы реализованы в основном в C и скомпилированы в собственный код, который запускается непосредственно на машине. Это контрастирует с обычными методами, которые реализованы в Java и скомпилированы в байт-код Java, который выполняется виртуальной машиной Java (JVM).

**какие плюсы и минусы**

минусы - скрыта реализация не можем воздействовать

**где хранятся ссылки на объекты?** - стек

**для чего созданы default методы в интерфейсе?**

при добавлении нового функционала в программу не делает необходимости в переопределении нового функционала

во всех участках где используется интерфейс

**что такое ранее и позднее связывание?**

ранее связывание - на этапе компиляции

позднее - на этапе выполнения

**что такое Enum?**

класс с конечным набором параметров

**методы Enum?**

values, ordinal, name

**Может ли Enum наследовать (implement) интерфейс в Java?**

Да, Enum может наследовать интерфейсы. Поскольку Enum тип схож с классом и интерфейсом, он может наследовать интерфейс. Это даёт поразительную гибкость в использовании Enum в качестве специальной реализации в некоторых случаях. Вот неплохой пример использования Enum в таком качестве.

**Может ли Enum наследовать (extends) класс?**

Нет, не может! Неожиданно, поскольку ранее говорилось что Enum тип похож на класс или интерфейс в Java. Ну, это главная причина, почему такой вопрос задают сразу за предыдущим. Поскольку Enum уже наследуется от абстрактного класса java.lang.Enum, понятно, что другой класс наследовать не удастся, поскольку Java не поддерживает множественное наследование классов. Благодаря наследованию от java.lang.Enum, все перечисления имеют методы ordinal(), values() или valueOf().

**требования к конструктору в Enum?** должен быть private или по умолчанию

**4. Какие модификаторы доступа могут быть у класса?**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Класс | Внутренний класс | Переменная | Метод | Конструктор | Логический блок |
| public | Да | Да (кроме локальных и анонимных классов) | Да | Да | Да | Нет |
| protected | Нет | Да (кроме локальных и анонимных классов) | Да | Да | Да | Нет |
| default | Да | Да | Да (и для локальной переменной) | Да | Да | Да |
| private | Нет | Да (кроме локальных и анонимных классов) | Да | Да | Да | Нет |
| final | Да | Да (кроме анонимных классов) | Да (и для локальной переменной) | Да | Нет | Нет |
| abstract | Да | Да (кроме анонимных классов) | Нет | Да | Нет | Нет |
| static | Нет | Да (кроме локальных и анонимных классов) | Да | Да | Нет | Да |
| native | Нет | Нет | Нет | Да | Нет | Нет |
| transient | Нет | Нет | Да | Нет | Нет | Нет |
| synchronized | Нет | Нет | Нет | Да | Нет | Да (только как часть метода) |
| volatile | Нет | Нет | Да | Нет | Нет | Нет |
| strictfp | Да | Да | Нет | Да | Нет | Нет |

**5. Почему класс верхнего уровня нельзя пометить модификатором static?**

Все классы верхнего уровня по определению статичны.

В чем сводится static, так это то, что экземпляр класса может стоять сам по себе. Или, наоборот: нестатический внутренний класс (= внутренний класс экземпляра) не может существовать без экземпляра внешнего класса. Поскольку класс верхнего уровня не имеет внешнего класса, он не может быть ничего, кроме static.

Поскольку все классы верхнего уровня являются статическими, наличие ключевого слова static в определении класса верхнего уровня бессмысленно.

**6. Как реализована неизменность String?  
7. Как реализовать свой Immutable тип данных?  
12. Синтаксические отличия интерфейса от абстрактного класса (АК)? 5 отличий**