**1. Какие “строковые” классы вы знаете?**

String, StringBuffer, StringBuilder

**2. Какие основные свойства “строковых” классов (их особенности)?**

**3. Чем отличаются и что общего у классов String, StringBuffer и StringBuilder?**

**4. Перечислите состав класса.**

**5. Где и как могут использоваться [static] [abstract] [final] в контексте класса?**

**6. Где могут использоваться слова super и this?**

**Super()** используется для обращения к базовому классу из дочернего.

**7. Для чего используется модификатор native?**

Java Native Interface (JNI) — стандартный механизм для запуска кода, под управлением виртуальной машины Java (JVM), который написан на языках С/С++ или Ассемблера, и скомпонован в виде динамических библиотек, позволяет не использовать статическое связывание. Это даёт возможность вызова функции С/С++ из программы на Java, и наоборот.

Модификатор native сигнализирует о том, что метод реализован в платформо-зависимом коде, часто на языке С.

Этот модификатор может быть применен только к методам, но не классам и переменным.

Тело нативного метода должно заканчиваться на (;) как в абстрактных методах, идентифицируя то, что реализация опущена.

**8. Что такое логический блок? Статический блок?**

Логический блок основная идея этого подхода состоит в том, что во всех случаях, когда требуется логически связать два или более оператора, это делается посредством создания блока.

Статические блоки вызываются в момент инициализации класса (когда ClassLoader будет грузить его в MetaSpace, [подробнее в JLS](http://docs.oracle.com/javase/specs/jls/se8/html/jls-12.html#jls-12.4.2)), и могут использоваться для иницализации статических переменных.

**9. Определите параметризованный класс.**

Обобщения - это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра.

Class Java<T> {

T ob;

}

**10. Как используется метасимвол «?»**

**11. Какие существуют generic-ограничения?**

**12. Для чего и как можно использовать лямбда выражения?**

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

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

**13. Когда можно опускать типы параметров в лямбда выражении?**

**14. Как задать ссылку на методм?**

**15. Что могут содержать перечисления? Приведите пример**

В простейшей форме перечисление - это список именованных констант. Но в Java перечисления имеют более сложный функционал, чем в других языках программирования. Они могут иметь конструкторы, методы и переменные экземпляра.

**16. Какие спецификаторы доступа могут быть у конструктора перечисления?**

**17. Какие существуют ограничения для перечисления?**

**18. Что такое методы подставки?**

С пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип.

*/\* пример # 5 : методы-подставки:* *CourseHelper.java:*

*BaseCourseHelper.java: RunnerCourse.java\*/*

**package** chapt04;

**public** **class** CourseHelper {

**public** Course getCourse(){

System.*out*.println("Course");

**return** **new** Course();

}

}

**package** chapt04;

**public** **class** BaseCourseHelper **extends** CourseHelper {

**public** BaseCourse getCourse(){

System.*out*.println("BaseCourse");

**return** **new** BaseCourse();

}

}

**package** chapt04;

**public** **class** RunnerCourse {

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

CourseHelper bch = **new** BaseCourseHelper();

Course course = bch.getCourse();

*//BaseCourse course = bch.getCourse();//ошибка компиляции*

System.*out*.println(bch.getCourse().id);

}

}

В данной ситуации при компиляции в подклассе **BaseCourseHelper** создаются два метода. При обращении к методу **getCourse()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении вызывается метод-подставка. Обращение к полю производится по типу ссылки, возвращаемой методом **getCourse()**, то есть к полю класса **Course**.

**19. Состав класса Object.**

Методы класса Object:  
public final native Class getClass()  
public native int hashCode()  
public boolean equals(Object obj)  
protected native Object clone() throws CloneNotSupportedException  
public String toString()  
public final native void notify()  
public final native void notifyAll()  
public final native void wait(long timeout) throws InterruptedException  
public final void wait(long timeout, int nanos) throws InterruptedException  
public final void wait() throws InterruptedException  
protected void finalize() throws Throwable

**20. Перечислите соглашения по equlas.**

1. Симметричность. Т.е. если для каких-либо объектов *x*и *y* x.equals(y) возвращает true, то и y.equals(x) должен возвращать true.
2. Рефлексивность. Для любого объекта *x*x.equals(x) должен возвращать true.
3. Постоянство. Для любых объектов *x* и *y*x.equals(y) возвращает одно и тоже, если информация, используемая в сравнениях, не меняется.
4. Транзитивность. Для любых объектов *x, y*и *z,*если x.equals(y) вернет true и y.equals(z) вернет true, то и x.equals(z) должен вернуть true.

**21. Перечислите соглашения по hashCode().**

* Всякий раз, когда метод вызывается у одного и того же объекта во время выполнения приложения, он должен возвращать одно и то же число, если используемая информация не изменяется. hashCode может возвращать разные значения для идентичных объектов в различных экземплярах приложения.
* Если два объекта равны, согласно equals, то их hashCode должны возвращать одинаковые значения.
* Обратное требование необязательно. Два неравных объекта могут возвращать одинаковый hashCode. Однако для повышения производительности, лучше, чтобы разные объекты возвращали разные коды.

**22. Перечислите соглашения по toString().**

**23. Что такое finalize? Зачем он нужен?**

Метод.

Предназначен этот метод для автоматического освобождения системных ресурсов, занимаемых объектом, на котором будет данный метод вызван. Это кажется удобным, чтобы не помнить постоянно, например, что мы должны закрыть соединение с каким-то ресурсом, когда оно больше не требуется.

**24. Поясните разницу между «неглубким» и «глубоким» клонированием? Приведите пример.**

**25. Как можно использовать метод void finalize()?**

**26. Что такое внутренние классы (inner)? Привила использования.**

**27. Что такое вложенные (nested) классы? Привила использования.**

**28. Что такое анонимные (anonymous) классы?**

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

Абстрактным называется класс, на основе которого не могут создаваться объекты. При этом наследники класса могут быть не абстрактными, на их основе объекты создавать, соответсвенно, можно.

**30. Можно ли перегрузить static метод?**

что вы НЕ можете переопределять (Override) статические методы.

**31. Правила определения и наследования интерфейсов.**

**32. Для чего используются статические методы в интерфейсе?**

1. Статические методы в интерфейсе являются частью интерфейса, мы не можем использовать его для объектов класса реализации.
2. Статические методы в интерфейсе хороши для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.
3. Статические методы в интерфейсе помогают обеспечивать безопасность, не позволяя классам, которые реализуют интерфейс, переопределить их.
4. Мы не можем определить статические методы для методов класса Object, потому что получим ошибку компиляции: «*This static method cannot hide the instance method from Object*«. Это потому, что в Java так делать нельзя  . То есть Object является базовым классом для всех классов и мы не можем использовать статический метод и еще такой метод с одинаковой сигнатурой.
5. Мы можем использовать статические методы интерфейса, чтобы не создавать вспомогательные классы, то есть переместить все статические методы в соответствующий интерфейс. Такой метод легко использовать и быстро находить.

**33. Как определить методы default в интерфейсе? Для чего?**

Для создания метода по умолчанию в интерфейсе, мы должны использовать ключевое слово default. Рассмотрим на простом примере создание метода по умолчанию.

Java



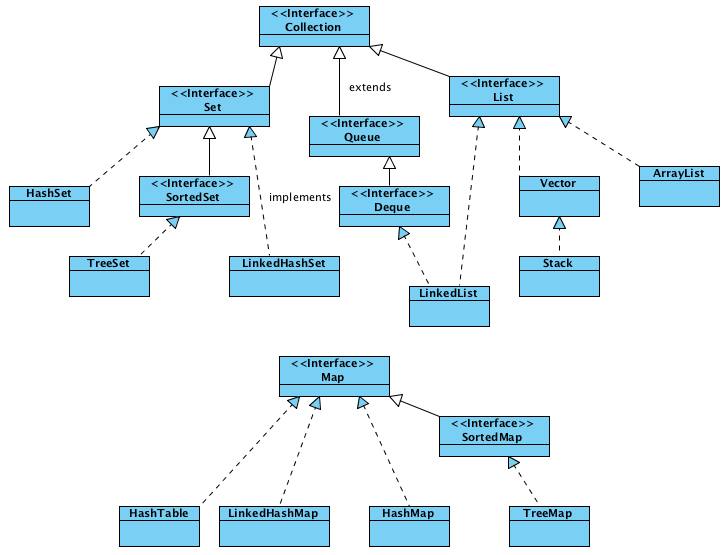
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public interface Interface1 {        void method1(String str);        default void log(String str){          System.out.println("Метод по умолчанию. Логгируем: " + str);          print(str);      }  } |

**34. Приведите иерархию исключений и ошибок? Поясните проверяемые и непроверяемые исключения.**

**35. Как написать собственное (“пользовательское”) исключение? Какими мотивами вы будете руководствоваться при выборе типа исключения: checked/unchecked?**

**36. В чем разница между интерфейсами Comparable и Comparator?**

**37. Какова иерархия коллекций?**



**38. Каково назначение коллекции java.util.Optional<T> ?**

**39. Перечислите встроенные аннотации.**

* **@Retention** - эта аннотация предназначена для применения только в качестве аннотации к другим аннотациям. Определяет политику удержания.
* **@Documented** - это маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована.
* **@Target** - эта аннотация задает тип объявления, к которым может быть применима аннотация. Принимает один аргумент, который должен быть константой из перечисления ElementType. Например, чтобы указать, что аннотация применима только к полям и локальным переменным: @Targer({ ElementType.FIELD, ElementTyle.LOCAL\_VARIABLE } )
* **@Inherited** - это аннотация-маркер, которая может применяться в другом объявление аннотации, она касается только тех аннотаций, что будут использованы в объявлениях классов. Эта аннотация позволяет аннотации супер класса быть унаследованной в подклассе.
* **@Override** - аннотация-маркер, которая может применяться только к методам. Метод, аннотированный как @Override, должен переопределять метод супер класса.
* **@Deprecated** - указывает, что объявление устарело и должно быть заменено более новой формой.
* **@SafeVarargs** - аннотация-маркер, применяется к методам и конструкторам. Она указывает, что никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Применяется только к методам и конструкторам с переменным количеством аргументов, которые объявлены как static или final.
* **@SuppressWarnings** - эта аннотация указывает, что одно или более предупреждений, которые могут быть выданы компилятором следует подавить.

**40. Как создать пользовательскую аннотацию?**

Аннотации создаются с использование механизма, основанного на интерфейсе. Ниже пример объявление аннотации:

@**interface** My{   
String str();   
**int** val();   
}

Символ @ - указывает компилятору, что объявлена аннотация. Методы, объявленные в аннотации ведут себя скорее как поля.

//Аннотирование метода.   
@My(str = "Пример аннотации", val = 100)   
**public** **static** **void** myMeth() { // ...