**Вопросы к лабораторной работе №3**

1.Перечислите состав класса.   
**[public] [final] [abstract] class ИмяКласса {** {}  
 // логические блоки   
// внутренние классы   
// поля   
private // закрытые   
public // открытые   
// дружественные (по умолчанию) пакетный уровень   
protected // защищенные   
// конструкторы   
public // открытые // дружественные (по умолчанию)   
protected // защищенные   
private // закрытые // методы   
public // открытые ...}

Рассмотрим основные элементы класса и их назначение:

– константы класса хранят неизменяемые значения;

– поля класса (типы и имена переменных класса);

– методы класса это поименованный фрагмент кода программы, предназначенный для работы с данными класса;

– свойства класса это совокупность методов, позволяющих классу обмениваться (читать или записывать) значениями полей класса с другими классами программы;

– конструкторы класса это специальные методы класса, которые предназначены для создания объектов класса и присваивания начальных значений полям класса;

– деструкторы класса это специальные методы, определяющие порядок действий при освобождении ресурсов, выделенных объекту;

– события класса это специальные методы, позволяющие классу реагировать на действия пользователя или на определенные изменения в программе;

– типы это типы данных, внутренние по отношению к классу. Например, перечисления, структуры, классы, делегаты, интерфейсы.

– индексаторы это средство доступа к элементам данных класса (обычно массивам) по их порядковому номеру;

–операции это специальные действия с объектами класса с помощью знаков операций.  
2. Где и как могут использоваться [static] [abstract] [final] в контексте класса?

**Модификаторы в Java** – это ключевые слова, которые придают классу, полю класса или методу определенные свойства.  
[Модификаторы классов](http://www.javenue.info/post/java-modifiers-summary#class-modifiers)  
Модификатор **abstract**, примененный к классу, говорит о том, что класс является (или считается) незаконченным, а задание "завершить" класс возлагается на наследников. Попытка инстанциировать такой класс приведет к ошибке компиляции.  
Модификатор **final** у класса говорит о том, что от него нельзя наследоваться.  
Вложенные классы в Java могут быть объявлены как **static**. В этом случае класс называется статическим вложенным классом и имеет доступ к статическим полям и методам обрамляющего класса

[Модификаторы методов](http://www.javenue.info/post/java-modifiers-summary#method-modifiers)

Метод с модификатором **abstract** может быть объявлен как метод-член в пределах абстрактного класса (или интерфейса). В этом случае тело метода отсутствует, а реализация может быть предоставлена в классах-наследниках. Если же метод объявлен как абстрактный в конкретном классе, то получим ошибку компиляции.  
Метод, объявленный с модификатором **final** не может быть переопределен в наследниках.  
Метод с модификатором **static** относится к классу в целом, а не к его экземплярам, то есть в него не передается объект this. Такой метод может быть вызван используя имя класса.

[Модификаторы полей](http://www.javenue.info/post/java-modifiers-summary#field-modifiers)  
Поля классов могут быть описаны с такими модификаторами как static, final, transient, volatile. Если поле класса объявлено как **static**, то будет существовать ровно одно значение этого поля, не зависимо от того, сколько экземпляров класса будет создано, даже если не будет создано ни одного экземпляра. Такие статические поля еще называют переменными уровня класса (class variable).

Поле с модификатором **final** не может поменять своего значения после инициализации. Это касается и статических, и нестатических полей (member fields).

[Модификаторы, связанные с интерфейсами](http://www.javenue.info/post/java-modifiers-summary#interface-modifiers)

К интерфейсам можно применить те же модификаторы, что и к классам, кроме final. Интерфейс является abstract по-умолчанию. В случае со вложенным интерфейсом ключевое слово static можно не указывать - он в любом случае будет статическим.  
Методы интерфейсов по-умлочанию являются public abstract, поэтому к ним не применимы модификаторы final, static и native. Синхронизированными они тоже быть не могут, так как интерфейс нельзя инстанциировать.

Поля интерфейсов по-умолчанию являются public static final, а значит должны быть проинициализированы.

[Другие контексты использования модификаторов](http://www.javenue.info/post/java-modifiers-summary#other-modifiers)  
локальные переменные, которые могут быть объявлены как final. В этом случае значение переменной нельзя будет изменить в пределах метода после инициализации

статический блок инициализации, который выполняется при загрузке класса:  
3. Где могут использоваться слова super и this?

**This** - Это ссылка на объект, типизированный как текущий класс, для вызова того же конструктора класса;  
**Super** - это ссылка на объект, указанный в качестве родительского класса. для вызова конструктора родительского класса.

**this**

Когда объект создается, JVM назначает указатель на объект, который ссылается на себя. Имя этого указателя называется this. Из-за этого это можно использовать только в нестатических методах. И это связано только с конкретным объектом, а не с классом.  
Есть четыре конкретных применения этого:  
1. Вызов переменных-членов в классе  
2. Вызов других методов построения в этом классе.  
3. Представляйте свои собственные объекты.  
4. Вызов метода участника

**Super**Super аналогичен этому. Это указатель на родительский класс, который может использоваться только в нестатических методах.  
1. Вызов метода или переменной-члена родительского класса.

4. Для чего используется модификатор native?

Native — это модификатор, который пишется перед объявлением метода. Он означает, что код метода написан не на Java, а на С++ и встроен в Java-машину (ну или подключаемую DLL-библиотеку).

5. Что такое логический и статический блок?

**Логическим блоком** называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса, например: {/\* код\*/} static {/\* код\*/}. Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов и обращения к полям текущего класса. **Логические блоки** вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, до вызова конструктора класса;

**Статический блок в** Java создаётся внутри класса с помощью ключевого слова static.

**Статический** **блок** используется для инициализации **статических** переменных.

Хотя **статические** переменные могут быть инициализированы непосредственно во время объявления, бывают ситуации, когда нам требуется выполнить многострочную обработку.  
6. Определите параметризованный класс.

**Параметризированный класс** – некоторый шаблон, на основе которого можно строить другие классы.

7. Как используется метасимвол «?»

**Метасимволы** — это символы, имеющие в регулярном выражении специальное значение.

Часто возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом. В этом случае используется метасимвол «?»  
**8. Какие существуют generic-ограничения?**

***Generic-ограничения***:

1) невозможно выполнить явный вызов конструктора generic-типа;

2) generic-поля не могут быть статическими;   
3) статические методы не могут иметь generic-параметры или обращаться к generic полям

9. Что могут содержать перечисления? Приведите пример   
Перечисления может содержать поля, конструкторы и методы, реализовывать интерфейсы  
10. Какие существуют ограничения для перечислений?

**Перечисления ограничения**: 1)конструкторы вызываются автоматически при инициализации - > не может public и protected 2)не может быть суперклассом 3)не может быть подклассами 4)не может быть абстрактными 5)не может создавать экземпляры, используя ключевое слово new

11. Что такое методы подставки?

При переопределении методов можно указывать другой тип возвращаемого значения – но только тип, находящийся ниже в иерархии наследования, чем исходный тип.

12. Состав класса Object.

[public] [final] [abstract] class ИмяКласса   
{   
{} // логические блоки   
// внутренние классы   
// поля   
private // закрытые   
public // открытые   
// дружественные (по умолчанию) пакетный уровень   
protected // защищенные // конструкторы   
public // открытые   
// дружественные (по умолчанию)   
protected // защищенные   
private // закрытые   
// методы   
public // открытые ...  
13. Перечислите соглашения по equlas:

►рефлексивность

►симметричность

►транзитивность

►непротиворечивость

► ненулевая ссылка при сравнении с литералом null всегда возвращает значение false  
14. Перечислите соглашения по hashCode() :

1) все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;

2) различные по содержанию объекты одного типа могут иметь различные хэш-коды;

3) во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен.  
15. Перечислите соглашения по toString(). :  
1) стандартная информация о пакете (опционально)   
2) имя класса (опционально)   
3) значения полей объекта  
16. Поясните разницу между «неглубким» и «глубоким» клонированием? Приведите пример. Когда вы делаете неглубокую копию, вы копируете ссылки на оригинал, а не делаете фактическую копию самих данных. Например, неглубокая копия списка строк не приведет к двум дублирующимся наборам данных. Вы не удвоите объем памяти. Вместо этого копия будет содержать собственные ссылки на исходный объект.  
Глубокая копия приведет к тому, что оба списка будут иметь два отдельных экземпляра  
17. Как можно использовать метод void finalize()?

Метод protected void finalize () — метод, который будет вызываться непосредственно перед окончательным уничтожением объекта сборщиком мусора. Метод может быть вообще не выполнен, если возникнет исключительная ситуация, она будет проигнорирована (финализатором). Фактически этот метод – противоположность конструктору. В нем можно освобождать ресурсы, используемые объектом.  
18. Что такое внутренние классы (inner)? Привила использования.   
**Внутренние классы** — когда объект внутреннего класса связан с объектом обрамляющего класса. Не статические вложенные классы называются внутренними классами, если они связанны с внешним классом. **внутренние** **классы** (**inner** class) - это **классы** определенные внутри другого **класс**

**Правила использования** :

1) Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса.

2)Внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса. Доступ будет разрешен по имени в том числе и к полям, объявленным как private.

3) Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (final static).

4)Внутренний класс может быть объявлен также внутри метода или логического блока внешнего (owner) класса. Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования.  
19. **Что такое вложенные (nested) классы? Правила использования.**

**Вложенные (nested) классы**-Статический внутренний класс логически связанный с классом владельцем называется вложенным (nested). Если класс вложен в интерфейс, то он становится статическим по умолчанию статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класс

**Правила использования**:

**Из него (самого класса) видны:**

1) статические свойства и методы OuterClassа (даже private).

2) статические свойства и методы родителя OuterClassа public и protected. То есть те, которые видны в OuterClassе.

**Из его экземпляра видны**:

— все (даже private) свойства и методы OuterClassа обычные и статические.

— public и protected свойства и методы родителя OuterClassа обычные и статические. То есть те, которые видны в OuterClassе.

**Его видно:** — согласно модификатору доступа.

**Может наследовать:**

— обычные классы.

— такие же статические внутренние классы в OuterClassе и его предках.  **Может быть наследован**:

— любым классом: — вложенным — не вложенным — статическим — не статическим!

**Может имплементировать интерфейс;**

**Может содержать:**

— статические свойства и методы.

— не статические свойства и методы. Для создания объекта вложенного класса объект внешнего класса создавать не надо.

20. Что такое анонимные (anonymous) классы?

Анонимный класс является разновидностью локального(объявляется только в блоке кода) класса без полноценного имени. Анонимно могут быть определены интерфейсы, перечисления и аннотации.

**Анонимные** внутренние **классы** — это  **классы**  без имени, которые предоставляют реализацию существующего **класса**.

**Анонимные (безымянные) классы применяются для :**

► придания уникальной функциональности отдельно взятому экземпляру,

► для обработки событий,

► реализации блоков прослушивания,

► реализации интерфейсов,

► запуска потоков,

► для реализации (переопределения) нескольких методов

► создания собственных методов объекта и т. д. Конструктор анонимного класса определить невозможно

Свойства

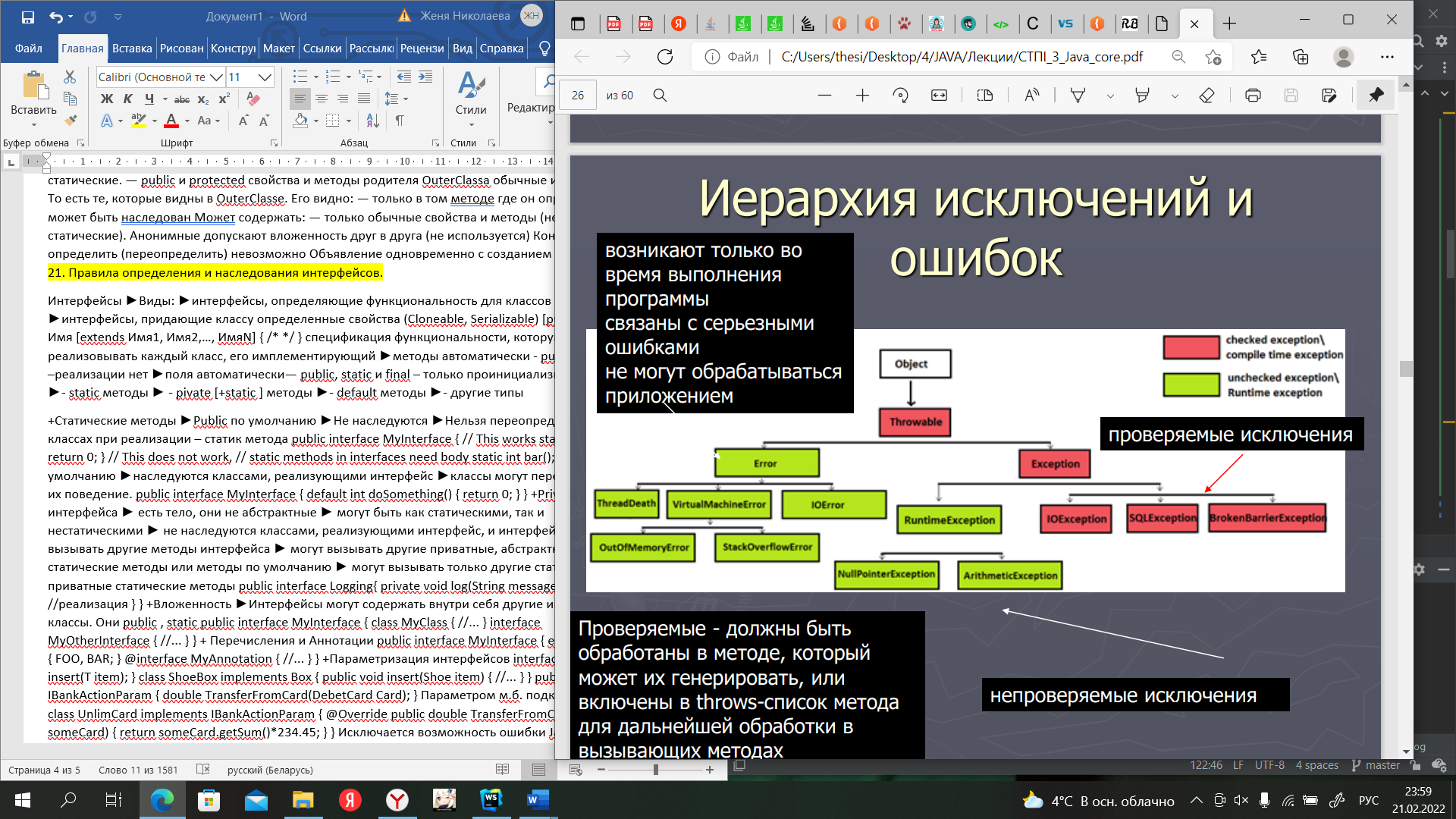
► **Из него видны**: — все (даже private) свойства и методы OuterClassа обычные и статические. — public и protected свойства и методы родителя OuterClassа обычные и статические. То есть те, которые видны в OuterClassе**. Его видно**: — только в том методе где он определён. Не может быть наследован.

**Может содержать**: — только обычные свойства и методы (не статические). Анонимные допускают вложенность друг в друга (не используется) Конструктор определить (переопределить) невозможно. Объявление одновременно с созданием объекта  
21. Правила определения и наследования интерфейсов.

**Интерфейсы**- спецификация функциональности, которую должен реализовывать каждый класс, его имплементирующий   
**Виды:** 1)интерфейсы, определяющие функциональность для классов; 2)интерфейсы, придающие классу определенные свойства (Cloneable, Serializable)   
[public] interface Имя [extends Имя1, Имя2,…, ИмяN] { /\* \*/ }

**методы автоматически** - public и abstract –реализации нет   
**поля автоматически**— public, static и final – только проинициализированные   
- static методы   
- pivate [+static ] методы  
- default методы  
- другие типы

**+Статические методы** :►Public по умолчанию ►Не наследуются ►Нельзя переопределять в классах при реализации – статик метода   
**+Методы по умолчанию** ►наследуются классами, реализующими интерфейс ►классы могут переопределять их поведение  
**+Private методы интерфейса** ► есть тело, они не абстрактные ► могут быть как статическими, так и нестатическими ► не наследуются классами, реализующими интерфейс, и интерфейсами ► могут вызывать другие методы интерфейса ► могут вызывать другие приватные, абстрактные, статические методы или методы по умолчанию ► могут вызывать только другие статические и приватные статические методы   
**+Вложенность** ►Интерфейсы могут содержать внутри себя другие интерфейсы и классы. Они public , static   
**+ Перечисления и Аннотации**   
**+Параметризация интерфейсов**   
**Функциональные интерфейсы** ► Компилятор проверяет, что аннотированная сущность представляет собой интерфейс с одним абстрактным методом ► Javadoc включает в себя утверждение, что ваш интерфейс является функциональным интерфейсом ► не обязаны использовать аннотацию ► Любой интерфейс с одним абстрактным методом является, по определению, функциональным интерфейсом.  
22. Приведите иерархию исключений и ошибок? Поясните проверяемые и непроверяемые исключения



Способы обработки исключений ►1) перехват и обработка исключения в блоке try-catch метода ►2) объявление исключения в секции throws метода и передача вызывающему методу ►3) использование собственных исключений контролируется компилятором   
1) Исключения – ресурсоемкие (если можно обойтись) 2) Можно создавать проверяемое исключение 3)не генерировать исключения в инструкции finally

Если тип сгенерированного исключения есть подклассом стандартного класса RuntimeException, то не обязательно указывать этот тип в перечне оператора throws метода. Такое исключение называется непроверяемым исключением. В этом случае компилятор не проверяет обрабатываются или генерируются такие исключения в некотором месте программы.

Если тип сгенерированного исключения не является подклассом стандартного класса RuntimeException, то это исключение называется проверяемое исключение. В случае генерирования такого типа исключения, его нужно обязательно включать в оператор throws.

В языке Java в пакете java.lang реализован ряд проверяемых исключений. Ниже приведен их перечень:

* ClassNotFoundException – класс не найден;
* CloneNotSupportedException – попытка клонировать объект из класса, который не реализует интерфейс Cloneable;
* IllegalAccessException – запрещен доступ к классу;
* InstantiationException – попытка создать объект абстрактного класса или интерфейса;
* InterruptedException – один поток выполнения прерван другим потоком;
* NoSuchFieldException – запрашиваемое поле не существует;
* NoSuchMethodException – запрашиваемый метод не существует;
* ReflectiveOperationException – суперкласс исключений, связанных с рефлексией.

Также, в перечень исключений оператора throws обязательно нужно включать собственноручно разработанные классы исключений для их проверки.