**1.Перечислите состав класса**.

**Класс** – логическое описание чего-либо, шаблон, с помощью которого можно создавать реальные экземпляры этого самого чего-либо. Другими словами, это просто описание того, какими должны быть созданные сущности: какими свойствами и методами должны обладать.

**Свойства** – характеристики сущности, методы – **действия**, которые она может выполнять.

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

Обозначая метод класса модификатором **final**, мы имеем в виду, что ни один производный класс не в состоянии переопределить этот метод, изменив его внутреннюю реализацию. Другими словами, речь идет о финальной версии метода. Класс в целом также может быть помечен как **final**.

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

Ключевое слово **super** в Java используется, когда подклассу требуется сослаться на его непосредственный суперкласс.

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Ключевое слово **this** в Java используется в теле любого метода для ссылки на текущий объект.

**4. 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.

**5. Что такое логический и статический блок?**

При описании класса могут быть использованы логические блоки. 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. При создании объекта класса они вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса. Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки **this**, представляющую собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static**. В этом случае он вызывается только один раз в жизненном цикле приложения при создании объекта или при обращении к статическому методу (полю) данного класса.

**6. Определите параметризованный класс.**

*Дженерики (или обобщения)* — это параметризованные типы.

Параметризованные типы позволяют объявлять классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Используя дженерики, можно создать единственный класс, например, который будет автоматически работать с разными типами данных.

Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются параметризованными или обобщениями, параметризованными (обобщенными) классами или параметризованными (обобщёнными) методами.

public class Matrix<T> {

private T[] array;

public Matrix(T[] array) {

this.array = array.clone();

}

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

**?**     - (знак вопроса) означает, что предшествующий ему символ является необязательным

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

Когда мы указываем универсальный параметр у обобщений, то по умолчанию он может представлять любой тип. Однако иногда необходимо, чтобы параметр соответствовал только некоторому ограниченному набору типов. В этом случае применяются ограничения, которые позволяют указать базовый класс, которому должен соответствовать параметр.

Для установки ограничения после универсального параметра ставится слово **extends**, после которого указывается базовый класс ограничения.

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

Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

enum Day{

     MONDAY,

    TUESDAY,

    WEDNESDAY,

    THURSDAY,

    FRIDAY,

    SATURDAY,

    SUNDAY

}

Перечисление как подкласс класса **Enum** может содержать поля, конструкторы и методы, реализовывать интерфейсы. Каждый тип **enum**может использовать методы:

**static enumType[]** **values()**– возвращает массив, содержащий все элементы перечисления в порядке их объявления;

**static T valueOf(Class<T> enumType, String arg)**– возвра­щает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки;

**static enumType** **valueOf(String arg)**– возвращает элемент пере­числения, соответствующий значению передаваемой строки;

**int ordinal()**– возвращает позицию элемента перечисления.

**10. Какие существуют ограничения для перечислений?**

Перечисления ограничения:

►конструкторы вызываются автоматически при инициализации -> не может public и protected

►не может быть суперклассом

►не может быть подклассами

►не может быть абстрактными

►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.**

В Java определен один специальный класс, называемый Object. Все остальные классы являются подклассами, производными от этого класса, даже если в объявлении это явно не указано. В классе Object определен ряд методов, которые доступны всем классам языка Java.

Методы класса Object в Java:

1. [protected Object clone()](http://www.examclouds.com/java/java-core-russian/cloning) - создает новый объект, не отличающийся от клонируемого.
2. [public boolean equals(Object obj)](http://www.examclouds.com/java/java-core-russian/method-equals) - определяет, равен ли один объект другому.
3. [protected void finalize()](http://www.examclouds.com/java/java-core-russian/metod-finalize) - вызывается перед удалением неиспользуемого объекта.
4. [public final Class<?> getClass()](http://www.examclouds.com/java/java-core-russian/class) - получает класс объекта во время выполнения.
5. public int hashCode() - возвращает хеш-код, связанный с вызывающим объектом.
6. public final void notify() - возобновляет исполнение потока, ожидающего вызывающего объекта.
7. public final void notifyAll() - возобновляет исполнение всех потоков, ожидающих вызывающего объекта.
8. [public String toString()](http://www.examclouds.com/java/java-core-russian/method-tostring) - возвращает символьную строку, описывающую объект.
9. public final void wait() - ожидает другого потока исполнения.
10. public final void wait(long timeout) - ожидает другого потока исполнения.
11. public final void wait(long timeout, int nanos) - ожидает другого потока исполнения.

**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 «неглубоким» и «глубоким» клонированием? Приведите пример.**

Лучший способ понять глубокое и мелкое клонирование — это:

1. В мелком клонировании если ссылочный объект присутствует он не будет сериализовать.
2. В глубоком клонировании как ссылка на объект, так и текущий оба сериализуются. Вместо этого с этим есть так много вещей, которые приходят в картину, как пул, ссылки на объекты и т. д

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

Если объект взаимодействует с какими-то ресурсами, например открывает поток вывода и читает из него, то такой поток необходимо закрыть перед удалением объекта из памяти. Для этого в языке Java достаточно переопределить метод finalize(), который вызывается в исполняющей среде Java непосредственно перед удалением объекта данного класса. В теле метода finalize() нужно указать те действия, которые должны быть выполнены перед уничтожением объекта. Метод finalize() вызывается лишь непосредственно перед сборкой "мусора”.

Метод finalize() не вызывается при выходе объекта из области действия. Заранее неизвестно, когда будет (и будет ли вообще) выполняться метод finalize(). И самое главное - начиная с Java 9 этот метод не рекомендуется к использованию.

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

***Особенности нестатических вложенных классов Java:***

1. Они существуют только у объектов, потому для их создания нужен объект. Другими словами: мы укомплектовали наше крыло так, чтобы оно было частью самолета, потому, чтобы создать крыло, нам нужен самолет, иначе оно нам не нужно.
2. Внутри Java класса не может быть статических переменных. Если вам нужны какие-то константы или что-либо еще статическое, выносить их нужно во внешний класс. Это связано с тесной связью нестатического вложенного класса с внешним классом.
3. У класса полный доступ ко всем приватным полям внешнего класса. Данная особенность работает в две стороны.
4. Можно получить ссылку на экземпляр внешнего класса.

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

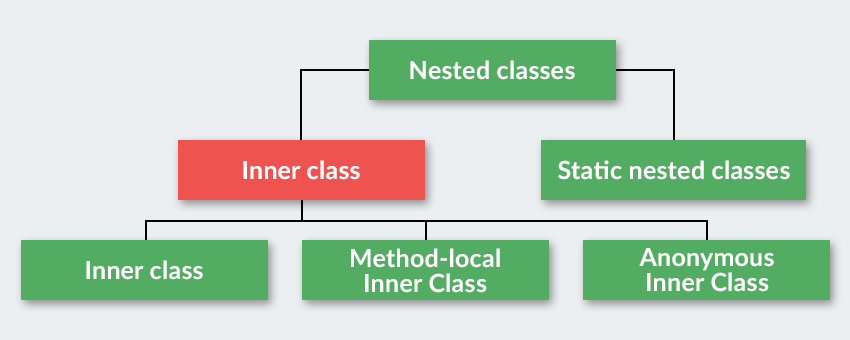
Java позволяет создавать одни классы внутри других

Именно такие классы и называют вложенными. Они делятся на 2 вида:

1. **Non-static nested** classes — **нестатические вложенные классы**. По-другому их еще называют **inner classes** — **внутренние классы**.
2. **Static nested** classes — **статические вложенные классы**.

В свою очередь, **внутренние классы (inner classes) имеют два особых подвида**. Помимо того, что внутренний класс может быть просто внутренним классом, он еще бывает:

* локальным классом (**local class**)
* анонимным классом (**anonymous class**)



**Внутренние классы** — это классы для выделения в программе некой сущности, которая неразрывно связана с другой сущностью.

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

Анонимный класс является разновидностью локального класса без

полноценного имени. Анонимно могут быть определены интерфейсы,

перечисления и аннотации.

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

**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)

Объявление одновременно с созданием объекта

**21. Правила определения и наследования 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.

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**

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

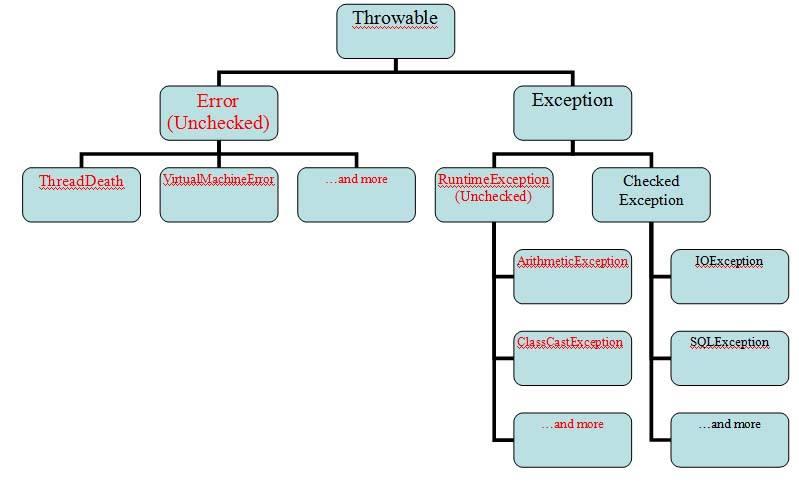
В мире программирования возникновение ошибок и непредвиденных ситуаций при выполнении программы называют **исключением**.

**Обработка исключений в Java основана на использовании в программе следующих ключевых слов:**

* **try** – определяет блок кода, в котором может произойти исключение;
* **catch** – определяет блок кода, в котором происходит обработка исключения;
* **finally** – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch, try{}catch{}finally, try{}finally{}.

* **throw** – используется для возбуждения исключения;
* **throws** – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.



Все исключения делятся на 2 вида — **проверяемые** (**checked**) и **непроверяемые** (**unchecked**).

Все проверяемые исключения происходят от класса **Exception**. Что значит “проверяемые”? Мы частично упоминали об этом [в прошлой лекции](https://javarush.ru/groups/posts/1943-iskljuchenija-perekhvat-i-obrabotka): **“...компилятор Java знает о самых распространенных исключениях, и знает, в каких ситуациях они могут возникнуть”.** Например, он знает, что если программист в коде считывает данные из файла, может легко возникнуть ситуация, что файл не существует. И таких ситуаций, которые он может заранее предположить, очень много. Поэтому компилятор заранее проверяет наш код на наличие потенциальных исключений. Если он их найдет, то не скомпилирует код, пока мы не обработаем их или не пробросим наверх.

Второй вид исключений — “непроверяемые”. Они происходят от класса **RuntimeException**. Чем же они отличаются от проверяемых? Казалось бы, тоже есть куча разных классов, которые происходят от **RuntimeException** и описывают конкретные runtime-исключения. Разница в том, что этих ошибок компилятор не ожидает. Он как бы говорит: “На момент написания кода я ничего подозрительного не обнаружил, но при его работе что-то пошло не так. Видимо, в коде есть ошибки!”