21. Зачем нужен оператор instanceof?  
22. Зачем нужны и какие бывают блоки инициализации?  
23. Каков порядок вызова конструкторов и блоков инициализации двух классов: потомка и его предка?  
24. Где и для чего используется модификатор abstract?  
25. Можно ли объявить метод абстрактным и статическим одновременно?  
26. Что означает ключевое слово static?  
27. К каким конструкциям Java применим модификатор static?  
28. Что будет, если в static блоке кода возникнет исключительная ситуация?  
29. Можно ли перегрузить static метод?  
30. Что такое статический класс, какие особенности его использования?  
31. Какие особенности инициализации final static переменных?  
32. Как влияет модификатор static на класс/метод/поле?  
33. О чем говорит ключевое слово final?  
34. Дайте определение понятию “интерфейс”.  
35. Какие модификаторы по умолчанию имеют поля и методы интерфейсов?  
36. Почему нельзя объявить метод интерфейса с модификатором final или static?  
37. Какие типы классов бывают в java (вложенные… и т.д.)  
38. Какие особенности создания вложенных классов: простых и статических.  
39. Что вы знаете о вложенных классах, зачем они используются? Классификация, варианты использования, о нарушении инкапсуляции.  
40. В чем разница вложенных и внутренних классов?  
41. Какие классы называются анонимными?  
42. Каким образом из вложенного класса получить доступ к полю внешнего класса?

**21. Зачем нужен оператор instanceof ?**

Оператор **instanceof** возвращает true, если объект является экземпляром класса или его потомком.

Оператор **instanceof** позволяет проверить, к какому классу принадлежит объект, с учётом наследования. Такая проверка может потребоваться во многих случаях. Здесь мы используем её для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

public class MainClass {

    public static void main(String[] a) {

    String **s** = "Hello";

    int **i** = 0;

    String **g**;

    if (**s** instanceof **java.lang.String**) {

       // попадем сюда, т.к. выражение будет true

       System.out.println("s is a String");

    }

    if (**i** instanceof **Integer**) {

       // instanceof НЕ РАБОТАЕТ С ПРИМИТИВАМИ!!!!! Ошибка компилятора!

       System.out.println("i is an Integer");

    }

    if (**g** instanceof **java.lang.String**) {

       // g не инициализирована и поэтому сюда мы не попадем, т.к.

       // g - null и instanceof вернет false для null.

       System.out.println("g is a String");

    }

}

------------------------------------------------------------------------

**Оператор instanceof нужен, чтобы проверить, был ли объект, на который ссылается переменная X, создан на основе какого-либо класса Y.**

Ты наверняка помнишь, что оператор instanceof возвращает значение true, если проверка показала истинность, или false, если результат был ложным. Следовательно, чаще всего он встречается в разного рода проверочных условиях (if…else).

public class Main {

public static void main(String[] args) {

Integer x = new Integer(22);

System.out.println(x instanceof Integer);

}

}

Как думаешь, что будет выведено в консоль? Ну, здесь это очевидно :) Объект **х** является Integer, поэтому результатом будет *true*.

Вывод в консоль: true

public class Animal {}

public class Cat extends Animal {}

public class MaineCoon extends Cat {}

Например, как думаешь, что выдаст вот такая проверка:

public class Main {

public static void main(String[] args) {

Cat cat = new Cat();

System.out.println(cat instanceof Animal);

System.out.println(cat instanceof MaineCoon);

}

}

Вывод: true false

Главный вопрос, на который нужно ответить, — как именно instanceof расшифровывает понятие «объект создан на основе класса»? У нас в результате получилось Сat instanceof Animal = true, но ведь к такой формулировке можно и придраться. Почему это объект Cat создается на основе класса Animal? Разве он не создается только на основе собственного класса? Ответ достаточно прост, и ты, возможно, уже додумался до него. Вспомни порядок вызова конструкторов и инициализации переменных при создании объекта.

**22. Зачем нужны и какие бывают блоки инициализации?**

Блоки инициализации представляют собой наборы выражений инициализации полей, заключенные в фигурные скобки и размещаемые внутри класса вне объявлений методов или конструкторов. Блок инициализации выполняется так же, как если бы он был расположен в верхней части тела любого конструктора. Если блоков инициализации несколько, они выполняются в порядке следования в тексте класса. Блок инициализации способен генерировать исключения, если их объявления перечислены в предложениях throws всех конструкторов класса.

Бывают статические и нестатические блоки инициализации. Также возможно создать такой блок в анонимном классе.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Foo {  static List<Character> abc;  static {  abc = new LinkedList<Character>();  for (char c = 'A'; c <= 'Z'; ++c) {  abc.add( c );  }  }  }  //Пример нестатического(ОН ЖЕ ЛОГИЧЕСКИЙ) блока инициализации  class Bar {  {  System.out.println("Bar: новый экземпляр");  }  }  //Пример инициализации в анонимном классе  JFrame frame = new JFrame() {  {  add(new JPanel() {  {  add(new JLabel("Хабрахабр?") {  {  setBackground(Color.BLACK);  setForeground(Color.WHITE);  }  });  }  });  }}; |

#### Виды блоков инициализации

Существует всего два типа блоков:

* **нестатический** (*instance initializer*)
* **статический**(*class initializer*)

В предыдущем примере Вы видели первый тип - нестатический блок инициализации. Его общую форму можно показать так:

Как можно догадаться по названию:

* **статический блок** используется для инициализации **статических переменных**, а "обычный"  - **для всех остальных**.

#### Зачем используются блоки инициализации

Красивый и читабельный код - это конечно хорошо. Но зачем они нужны? Есть ли другие преимущества у блоков инициализации? **Есть.** И это - **больший функционал**.

Внутри блоков инициализации мы можем не только присваивать значения. Это **как метод** - тут можно писать любые команды. Например, вывод в консоль:

|  |  |
| --- | --- |
|  | class Test{  **static {**  **System.out.println("Это статический блок!");**  **}**    {              System.out.println("Это нестатический блок!");  }   } |

#### Подводим итог:

* блоки инициализации используются для инициализации переменных внутри класса;
* есть два типа блоков инициализации - статический и нестатический;
* статический блок инициализации используется для статических переменных, нестатический - для всех остальных;
* блоки инициализации делают код читабельнее, и позволяют вызывать любые методы.

#### 23. Каков порядок вызова конструкторов и блоков инициализации двух классов: потомка и его предка?

Поля объекта инициализируются в следующем порядке:

1. Статические поля класса Parent;
2. Статический блок инициализации класса Parent;
3. Статические поля класса Сhild;
4. Статический блок инициализации класса Child;
5. Заходим в конструктор Child и первое, что вызывается – super(), т.е. конструктор Parent, т.о. МЫ СРАЗУ ЖЕ ВЫХОДИМ ИЗ КОНСТРУКТОРА Child!
6. Нестатические поля класса Parent инициализируются default-значениями, либо действительными(если они указаны!);
7. Нестатический блок инициализации класса Parent;
8. Конструктор класса Parent;
9. Нестатические поля класса Сhild(аналогично с Parent!);
10. Нестатический блок инициализации класса Сhild;
11. Тут ВОЗВРАЩАЕМСЯ В Конструктор класса Сhild!

Pet cat = new Cat("Barsik");

При вызове **new**, проверяется проинициализированы ли статические поля и блоки указанного типа(Cat).

Если проинициализирована, то вся статическая часть(поля и блоки) пропускается.

Если нет, инициализируются в порядке, указанном выше.

Затем заходит в скобки(вызов конструктора) и см.всю цепочку выше!

public class Pet {  
 public static final String *DEFAULT\_KIND* = "dog";  
 public static final String *FIELD\_REGEX* = "\\w+";  
 private static final Pattern *kindPattern*;  
 private String kind;  
  
 static {  
 *kindPattern* = Pattern.*compile*(*FIELD\_REGEX*);  
 }  
  
 {  
 kind = *DEFAULT\_KIND*;  
 }  
  
 public Pet(String kind) {  
 Matcher matcher = *kindPattern*.matcher(kind);  
 if (matcher.matches()) this.kind = kind;  
 }  
  
 public String getKind() {  
 return kind;  
 }  
}

public class Cat extends Pet {  
  
 public static final String *DEFAULT\_NAME* = "Murka";  
 public static final Pattern *namePattern*;  
 String name;  
  
 static {  
 *namePattern* = Pattern.*compile*(*FIELD\_REGEX*);  
 }  
  
 {  
 name = *DEFAULT\_NAME*;  
 }  
  
 public Cat(String name) {  
 super("cat");  
 Matcher matcher = *namePattern*.matcher(name);  
 if (matcher.matches()) this.name = name;  
 }

}

#### 24. Где и для чего используется модификатор abstract?

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, например:

**public** **abstract** **class** Expression {

Expression e = **new** Expression();

}

В результате компиляции этого кода получим ошибку: Expression is abstract; cannot be instantiated.

Обратите внимание, у абстрактного класса не обязательно должны быть абстрактные методы.

#### Метод с модификатором ****abstract**** может быть объявлен как метод-член (member method) 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 не в абстрактном, то получим ошибку компиляции.

#### \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Создать экземпляр абстрактного класса невозможно! При этом наследники класса могут быть и абстрактными.

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.

Абстрактный метод — метод, который не имеет реализации. Если в классе есть хотя бы один абстрактный метод, то весь класс должен быть объявлен абстрактным.

Абстрактные методы могут объявлены в ENUM, в случае если мы хотим обязать его экземпляры реализовывать данные методы!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public abstract class Fighter {      abstract void fight();  }    public class JudoFighter extends Fighter {  *@Override*      void fight() {          System.out.println("Учу ушу, руками машу! Бью с лету в душу...");      }  } |

Использование абстрактных классов и методов позволяет описать некую абстракцию, которая должна быть реализована в других классах. Например, мы можем создать абстрактный класс Fighter и объявить в нём абстрактный метод fight(). Т.к. стилей борьбы может быть много, то, например, для JudoFighter extends Fighter метод fight() будет описывать приемы в стиле дзюдо и т.д.

#### 25. Можно ли объявить метод абстрактным и статическим одновременно?

**Нет**. Получите: Illegal combination of modifiers: ‘abstract’ and ‘static’. Модификатор abstract говорит, что метод будет реализован в другом классе, а static-метод обязан иметь реализацию и работать независимо от контекста конкретного экземпляра!

#### 26. Что означает ключевое слово static?

Модификатор static говорит о том, что метод или поле класса принадлежат не объекту, а классу. Т.е. доступ можно будет получить и не создавая объекта класса. Поля помеченные static инициализируются при инициализации класса. К примеру, Class.forName(«MyClass», true, currentClassLoader), где второй параметр указывает на необходимость проведения инициализации.

На методы, объявленные как static, накладывается ряд ограничений.

* Они могут вызывать только другие статические методы.
* Они должны осуществлять доступ только к статическим переменным.
* Они не могут ссылаться на члены типа this или super.

#### 27. К каким конструкциям Java применим модификатор static?

* полям;
* методам;
* вложенным классам;
* членам секции import (с 5-ой java). Например, import static org.junit.Assert.assertThat;

#### 28. Что будет, если в static блоке кода возникнет исключительная ситуация?

Если в явном виде написать любое исключение в static-блоке, то компилятор не скомпилирует исходники. Это все от того, что компилятор умный. В остальном, взаимодействие с исключениями такое же как и в любом другом месте. Если unchecked исключение вывалится в static-блоке, то класс не будет инициализирован.  
**Какое исключение выбрасывается при ошибке в блоке инициализации?**

Для static: java.lang.ExceptionInInitializerError — если исключение наследуется от RuntimeException.

Для init: exception, который и вызвал исключение, если он наследуется от RuntimeException.

Верно для static и init: java.lang.Error — если исключение вызвано Error.

java.lang.ThreadDeath — смерть потока. Ничего не вываливается.

#### 29. Можно ли перегрузить static метод?

Перегрузить можно, но переопределить нельзя.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public Animal eat() {          System.out.println("animal eat");          return null;      }      public static Animal eat(String s) {          System.out.println("test static overload");          return null;      } |

#### -------------------------------------------------------------------------------

На перегрузку (**overload**) статического метода не накладывается никаких ограничений. С точки зрения компилятора, методы с разным списком аргументов – разные методы. Но это не переопределение.  
  
Метод с [модификатором static](https://itsobes.ru/JavaSobes/chto-takoe-static) относится к классу, а не к его объектам. Для него работает статическое связывание, поэтому именно переопределение (**override**) в дочернем классе не работает.  
  
Несмотря на это, в дочернем классе можно объявить static метод с такой же сигнатурой, как в родительском. В этом случае произойдет не перегрузка и не переопределение, а перекрытие (**shadowing**). К такому методу нельзя применить аннотацию @Override, в нём нельзя использовать ключевое слово [super](https://itsobes.ru/JavaSobes/dlia-chego-nuzhno-kliuchevoe-slovo-super).  
  
*Если вы вызываете статический метод от переменной, а не типа, перекрытие таит в себе опасность. Без динамического связывания компилятор знает только о типе переменной, но не о типе ее значения. Если объявленный тип переменной – базовый класс, то метод-перекрытие никогда не вызовется. Поэтому при попытке такого вызова в IDE мы видим предупреждение.*

#### <https://itsobes.ru/assets/JavaSobes/262.jpg>

#### 30. Что такое *статический класс*, какие особенности его использования?

Это вложенный класс, который может обращаться только к статическим полям обертывающего его класса, в том числе и приватным. Доступ к нестатическим полям обрамляющего класса может быть осуществлен только через ссылку на экземпляр обрамляющего объекта. **К** **классу высшего уровня** модификатор static **неприменим**.

В примере показано, что для инициализации внутреннего статического класса нет нужды в инициализации родителя. Но в случае обычного внутреннего класса такой номер не пройдет:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class Test {    class A { }    static class B { }    public static void main(String[] args) {      /\*will fail - compilation error, you need an instance of Test to instantiate A\*/      A a = new A();      /\*will compile successfully, no instance of Test is needed to instantiate B \*/      B b = new B();    }  } |

Статические вложенные классы, не имеют доступа к нестатическим полям и методам обрамляющего класса, что в некотором роде аналогично статическим методам, объявленным внутри класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр обрамляющего класса. В этом плане static nested классы очень похожи на любые другие классы верхнего уровня.

#### 31. Какие особенности инициализации final static переменных?

Переменные должны быть инициализированы во время объявления или в static блоке.

Private static final гарантирует, что этот экземпляр не подменится на что-то другое.

Удобно при работе с базами данных или каким-то ресурсом, не склонным к разделению.

#### 32. Как влияет модификатор static на класс/метод/поле?

Модификатор static говорит о том, что метод или поле класса принадлежат не объекту, а классу.

Внутри static метода нельзя вызвать не статический метод по имени класса.

*МЕТОДЫ*: Статические методы также называются методами класса, потому что статический метод принадлежит классу, а не его объекту. Кроме того, статические методы можно вызывать напрямую через имя класса.

*ПЕРЕРМЕННЫЕ:* Если мы объявим переменную статической, все объекты класса будут использовать одну и ту же статическую переменную.

СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ  —  РЕДКОСТЬ В JAVA. ВМЕСТО НИХ ПРИМЕНЯЮТ СТАТИЧЕСКИЕ КОНСТАНТЫ. ОНИ ОПРЕДЕЛЯЮТСЯ КЛЮЧЕВЫМ СЛОВОМ STATIC FINAL И ПРЕДСТАВЛЕНЫ В ВЕРХНЕМ РЕГИСТРЕ. ВОТ ПОЧЕМУ НЕКОТОРЫЕ ПРЕДПОЧИТАЮТ ИСПОЛЬЗОВАТЬ ВЕРХНИЙ РЕГИСТР И ДЛЯ СТАТИЧЕСКИХ ПЕРЕМЕННЫХ.

*КЛАССЫ:* Про static класс (ответ №30).

#### 33. О чем говорит ключевое слово final?

Может быть применено к полям, методам или классам. В зависимости к какой сущности приложено данное ключевое слово — будет и различный смысл в его применении.

* Для класса. Класс помеченный при помощи final не может иметь наследников.
* Для метода. Метод помеченный при помощи final не может быть переопределен в классах наследниках.
* Для поля. Поле, помеченное при помощи слова final не может изменить свое значение после инициализации (инициализируется либо при описании, либо в конструкторе, статическом или динамическом блоке).
* Значение локальных переменных, а также параметров метода помеченных при помощи слова final не могут быть изменены после присвоения. !!!! final  для локальных переменных – bad practice!!!!!

#### 34. Дайте определение понятию “интерфейс”.

#### interface – это контракт, обязывающий класс реализовывать определённое поведение. При этом один класс может реализовывать несколько контрактов.

Ключевое слово **interface** используется для описания поведения, требующего дальнейшей реализации в классах. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов.

Интерфейс может содержать также:

* Default-методы - имеют реализацию по умолчанию (имеют тело) и НЕ ТРЕБУЮТ ПРЕРЕОПРЕДЕЛЕНИЯ.
* Static-методы – имеют тело
* Константы. Они public static final по умолчанию.

|  |  |
| --- | --- |
| 1  2  3  4 | public interface SomeName{      void method();      int getSum();  } |

#### 35. Какие модификаторы по умолчанию имеют поля и методы интерфейсов?

Интерфейс может содержать поля, но они автоматически являются статическими (static) и неизменными (final). Все методы и переменные неявно объявляются как public.

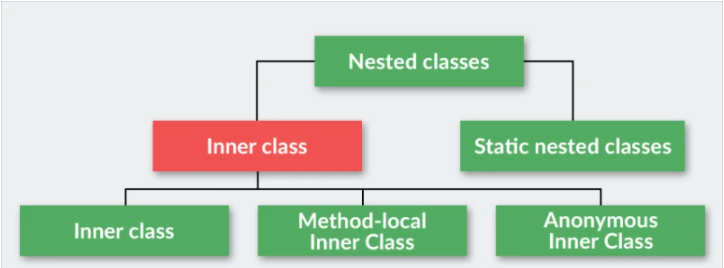
#### 36. Почему нельзя объявить метод интерфейса с модификатором final или static?

Вообще с 8-й версии можно static, но нужно чтобы было тело метода. Например

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface Shape {      static void draw() {          System.out.println("Wow! It is impossible!");      };  } |

**final** модификатор просто бессмысленный. Все методы по умолчанию абстрактные, т.е. их невозможно создать не реализовав где-то еще, но это нельзя будет сделать, если у метода идентификатор final.

#### 37. Какие типы классов бывают в java (вложенные… и.т.д.)



В Java есть 4 вида классов внутри другого класса:

1. ***Вложенные внутренние классы****– нестатические классы внутри внешнего класса.*
2. ***Вложенные статические классы****– статические классы внутри внешнего класса.*
3. ***Локальные классы Java****– классы внутри методов.*
4. ***Анонимные Java классы****– классы, которые создаются на ходу.*

Про каждый из них будем говорить отдельно.

### *Нестатические классы* внутри внешнего класса

Сначала, я хочу, чтобы вы осознали, что это такое, на реальном примере, потому что это облегчает понимание в разы. Так что сейчас мы будем разбивать реальную большую вещь на более мелкие составные части, а разбирать мы будем – самолёт! Однако, для примера будет достаточно показать немного, полностью разбивать мы не будем. Для визуализации данного процесса, будем использовать схему самолёта. Для начала, нам нужно создать класс Airplane, куда мы можем занести немного описание: название самолета, идентификационный код, рейс.

**public** **class** Airplane {

**private** String name, id, flight;

**public** Airplane(String name, String id, String flight) {

**this**.name = name;

**this**.id = id;

**this**.flight = flight;

}

// getters/setters

}

Теперь мы хотим добавить крылья. Создавать отдельный класс? Возможно в этом и есть логика, если у нас сложная программа для конструирования самолетов, и где нам нужно создавать огромное количество производных классов (классы, которые обладают такой же логикой, как и родительский класс, то есть класс, от которого они наследуются, но так же расширяют родительский класс, добавляя логику или более подробные характеристики), но что, если у нас просто игра, где у нас есть один самолет? Тогда нам будет рациональней укомплектовать всю структуру в одном месте (в одном классе). Тут идут в бой нестатические вложенные классы. По сути, это более подробное описание каких-то деталей нашего внешнего класса. В данном примере, нам нужно создать крылья для самолета – левое и правое. Давайте создавать!

**public** **class** Airplane {

**private** String name, id, flight;

**private** Wing leftWing = **new** Wing("Red", "X3"), rightWing = **new** Wing("Blue", "X3");

**public** Airplane(String name, String id, String flight) {

**this**.name = name;

**this**.id = id;

**this**.flight = flight;

}

**private** **class** Wing {

**private** String color, model;

**private** Wing(String color, String model) {

**this**.color = color;

**this**.model = model;

}

// getters/setters

}

// getters/setters

}

Так мы создали нестатический вложенный класс Wing (крыло) внутри класса Airplane (самолет), и добавили две переменные – левое крыло и правое крыло. И у каждого крыла есть свои свойства (цвет, модель), которые мы можем изменять. Так можно укомплектовывать структуры столько, сколько нужно. И заметьте: ранее на схеме было довольно много деталей у самолета, и, по сути, мы можем все детали разбить на внутренние классы, однако не всегда такой процесс целесообразен. Такие моменты нужно прослеживать в зависимости от задачи. Возможно, вам вообще не нужны крылья для решения задачи. Тогда и незачем их делать. Это как распилить человека на ноги, руки, торс и голову – можно, но зачем, если данный класс используется только для хранения данных об людях?

***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. Пример: Airplane.this – ссылка на самолет, this – ссылка на крыло.

### *Статические классы* внутри внешнего класса

Данный вид классов не отличается ничем от обычного внешнего класса, кроме одного: для создания экземпляра такого класса, нужно через точку перечислить весь путь от внешнего класса до нужного. Например: Building.Plaftorm platform = new Building.Platform(); Статические классы используются для того, чтобы укомплектовать связанные классы рядышком, чтобы с логической структурой было работать проще. Например: мы можем создать внешний класс Building, где будет конкретный список классов, которые будут представлять из себя уже конкретную постройку.

**public** **abstract** **class** Building {

**private** String name, address, type;

Building(String name, String address) {

**this**.name = name;

**this**.address = address;

}

**public** **static** **class** Platform **extends** Building {

**public** Platform(String name, String address) {

**super**(name, address);

setType("Platform");

}

// some additional logic

}

**public** **static** **class** House **extends** Building {

**public** House(String name, String address) {

**super**(name, address);

setType("House");

}

// some additional logic

}

**public** **static** **class** Shop **extends** Building {

**public** Shop(String name, String address) {

**super**(name, address);

setType("Shop");

}

// some additional logic

}

// getters/setters

}

Данный пример демонстрирует, как статические классы позволяют укомплектовывать логическую структуру в более удобный вид. Если бы их не было, нам бы понадобилось создавать 4 совершенно разных класса. ***Плюсы такого подхода:***

1. Количество классов уменьшилось.
2. Все классы внутри их класса-родителя. Мы способны прослеживать всю иерархию без открытия каждого класса отдельно.
3. Мы можем обратиться к классу Building, а IDE уже будет подсказывать весь список всех подклассов данного класса. Это будет упрощать поиск нужных классов и показывать всю картину более цело.

***Пример создания экземпляра вложенного статического класса:***Building.Shop myShop = new Building.Shop(“Food & Fun!”, “Kalyaeva 8/53”); Хотелось бы еще отметить, что данная стратегия задействована в 2D классах AWT для описания фигур, таких, как Line2D, Arc2D, Ellipse2D и другие.

### *Локальные классы*

Данные классы объявляются внутри других методов. По сути, они обладают всеми свойствами нестатического вложенного класса, только создавать их экземпляры можно только в методе, при чем метод не может быть статическим (для их создания нужен экземпляр внешнего класса, в нестатические методы неявно передается ссылка на экземпляр вызывающего объекта, а в статическом методе данной ссылки нет). ***Но, свои особенности у них есть:***

1. Локальные классы способны работать только с final переменными метода. Все дело в том, что экземпляры локальных классов способны сохраняться в «куче» после завершения работы метода, а переменная может быть стёрта. Если же переменная объявлена final, то компилятор может сохранить копию переменной для дальнейшего использования объектом. И еще: с 8+ версий Java можно использовать не final переменные в локальных классах, но только при условии, что они не будут изменяться.
2. Локальные классы нельзя объявлять с модификаторами доступа.
3. Локальные классы обладают доступом к переменным метода.

Локальные классы можно встретить крайне редко, так как они затрудняют прочтение кода и не обладают никакими плюсами, кроме одного – доступ к переменным метода. Я не знаю, какой можно взять пример локального класса, который бы показал их эффективное применение, так что покажу просто свой пример. Допустим, что у нас есть класс Person (будет считать, что это человек) со свойствами street (улица), house (дом). Нам бы хотелось возвращать какой-то объект для доступа только к местоположению человека. Для этого, мы создали интерфейс AddressContainer, который подразумевает собой хранилище данных об местоположении человека.

**public** **class** Person {

**private** String name, street, house;

**public** Person(String name, String street, String house) {

**this**.name = name;

**this**.street = street;

**this**.house = house;

}

**private** **interface** AddressContainer {

String getStreet();

String getHouse();

}

**public** AddressContainer getAddressContainer() {

**class** PersonAddressContainer **implements** AddressContainer {

**final** String street = Person.**this**.street, house = Person.**this**.house;

@Override

**public** String getStreet() {

**return** **this**.street;

}

@Override

**public** String getHouse() {

**return** **this**.house;

}

}

**return** **new** PersonAddressContainer();

}

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

Person person = **new** Person("Nikita", "Sholohova", "17");

AddressContainer address = person.getAddressContainer();

System.out.println("Address: street - " + address.getStreet() + ", house - " + address.getHouse());

}

// getters/setters

}

Как можно заметить, внутри метода мы создали класс, реализующий хранилище местоположения человека, создали там константные переменные (чтобы после выхода из метода переменные хранились в объекте) и реализовали метод для получения адреса и дома. Теперь мы можем использовать данный объект в других местах программы, чтобы получать местоположение человека. Понимаю, что данный пример неидеальный и его было правильней сделать просто оставив геттеры в классе Person, однако создание данного класса и его возможное использование было показано, а далее решать вам.

### *Анонимные классы*

Под капотом анонимные классы – просто обычные нестатические вложенные классы. Их особенность в удобстве их использования. Вы можете написать свой класс прямо при создании экземпляра другого класса.

**public** **class** Animal {

**public** **void** meow() {

System.out.println("Meow!");

}

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

Animal anonTiger = **new** Animal() {

@Override

**public** **void** meow() {

System.out.println("Raaar!");

}

};

Animal notAnonTiger = **new** Animal().**new** Tiger();

anonTiger.meow(); // будет выведено Raaar!

notAnonTiger.meow(); // будет выведено Raaar!

}

**private** **class** Tiger **extends** Animal {

@Override

**public** **void** meow() {

System.out.println("Raaar!");

}

}

}

По сути, мы просто совмещаем в одном месте две вещи: создание экземпляра одного класса (Animal) и создание экземпляра его внутреннего-класса наследника (Tiger). Иначе нам нужно создавать класс отдельно и использовать более длинные конструкции, чтобы добиться того же самого результата. ***Использование анонимных классов оправдано во многих случаях, в частности когда:***

* тело класса является очень коротким;
* нужен только один экземпляр класса;
* класс используется в месте его создания или сразу после него;
* имя класса не важно и не облегчает понимание кода.

Часто анонимные классы используются в графических интерфейсах для создания обработчиков событий. Например для создания кнопки и реакции на её нажатие:

JButton b2 = **new** JButton("Click");

b2.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

System.out.println("Кнопка нажата!");

}

});

Однако после Java 8 начали использовать лямбда-выражения, но все равно много кода было написано до 8 версии и вы можете столкнуться (и столкнетесь в ходе обучения на JavaRush) с такими вот надписями.\ Аналог с лямбдами:

JButton b2 = **new** JButton("Click");

b2.addActionListener(e -> System.out.println("Кнопка нажата!"));

#### 38. Какие особенности создания вложенных классов: простых и статических.

Если в **нестатическом классе** **объявляется вложенный статический** **класс**, то:

* для того, чтобы создать объект вложенного статического класса не нужно создавать объект охватывающего класса. Доступ к имени вложенного статического класса с именем InClass осуществляется по образцу OutClass.InClass, где OutClass – имя охватывающего класса;
* нельзя обращаться к членам охватывающего класса из статического вложенного класса.
* Для получения ссылки на объект внешнего класса используется конструкция вида

*OutClass.****this***

где OutClass – имя внешнего (охватывающего) класса. Конструкция .this подходит **только для вложенных нестатических классов**.

#### -------------------------------------------------------------

* Обычные классы (Top level classes)
* Интерфейсы (Interfaces)
* Перечисления (Enum)
* Статические вложенные классы (Static nested classes)
  + Есть возможность обращения к внутренним статическим полям и методам класса обертки.
  + Внутренние статические классы могут содержать только статические методы.
* Внутренние классы-члены (Member inner classes)
  + Есть возможность обращения к внутренним полям и методам класса обертки.
  + Не может иметь статических объявлений.
  + Нельзя объявить таким образом интерфейс. А если его объявить без идентификатора static, то он автоматически будет добавлен.
  + Внутри такого класса нельзя объявить перечисления.
  + Если нужно явно получить this внешнего класса — OuterClass.this
* Локальный класс (Local inner classes)
  + 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).
  + Имеют доступ к полям и методам обрамляющего класса.
  + Можно обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final.
* Анонимные классы (Anonymous inner classes)
  + Локальный класс без имени.

#### 39. Что вы знаете о вложенных классах, зачем они используются? Классификация, варианты использования, о нарушении инкапсуляции.

#### См.ответ 37

* Статические вложенные классы (Static nested classes)
  + Есть возможность обращения к внутренним статическим полям и методам класса обертки.
  + Внутренние статические классы могут содержать только статические методы.
* Внутренние классы-члены (Member inner classes)
  + Есть возможность обращения к внутренним полям и методам класса обертки.
  + Не может иметь статических объявлений.
  + Нельзя объявить таким образом интерфейс. А если его объявить без идентификатора static, то он автоматически будет добавлен.
  + Внутри такого класса нельзя объявить перечисления.
  + Если нужно явно получить this внешнего класса — OuterClass.this

#### 40. В чем разница вложенных и внутренних классов?

1. **Вложенные** классы(**nested**), объявленные статически, называются вложенными статическими классами. Имеют доступ только к статическим полям и методам внешнего класса.
2. **Внутренние** классы(**inner**) — когда объект внутреннего класса связан с объектом обрамляющего класса. Не статические вложенные классы называются внутренними классами, если они связанны с внешним классом. Имеют доступ к статическим и не статическим полям и методам внешнего класса.
3. Разница в создании объектов(вызов конструктора):

* Nested - new OutClassName.NestedClassName()
* Inner - new OutClassName(). new InnerClassName();

#### 41. Какие классы называются анонимными?

**Анонимный класс (anonymous class)** — это локальный класс без имени(он виден только в рамках метода, где был создан!).

**anonymous class** имеет только один экземпляр, т.к. этот класс не имеет имени => не может быть вызван его конструктор!

Class OutClass{

void doSmth(){

new InterfaceName {

void doSmth(){ }

}

}

}

#### 42. Каким образом из вложенного класса получить доступ к полю внешнего класса?

Если вложенный класс не статический(inner) и поле не статическое, то можно просто обратиться к этому полю из внутреннего класса.

Если у внутреннего класса(inner) не существует поля с таким же именем, в этом случае нужно обращаться через ссылку на внешний класс так — OuterClass.this.имяПоля

\*Из комментария к статье: достаточно, чтобы класс был не статическим, а поле статическое или нет — значения не имеет для Member Inner Class.

Если же вложенный класс статический(nested), данную операцию выполнить невозможно, т.к. nested класс привязан лишь к имени внешнего класса, но не к его экземпляру !!!!!!