**Компоненты Java программы:**

*Класс* – способ описания сущности, описывает его состояние, поведение и тип, а также правила взаимодействия с данной сущностью (контракт класса – комментарий разработчика, о том, какое поведение ожидается от класса).

*Компоненты класса:*

Поля (static, non-static), конструкторы, методы (static, non-static), блоки инициализации (static, non-static).

*Объект* – Конкретный представитель класса, имеющий конкретное состояние (определяемое конструктором и полями класса) и поведение (определяемое методами), полностью определяемое классом.

*Интерфейс* – контракт, определяющий набор методов, которые класс должен реализовать.

Интерфейс класса – набор публичных полей и методов, доступных к использованию другими классами.

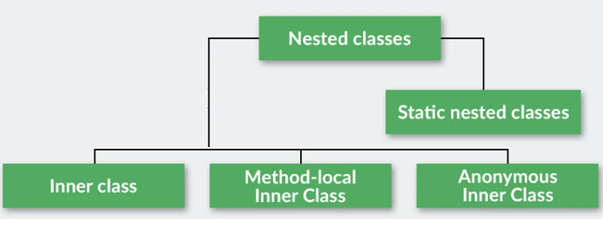
*Функциональный интерфейс* – интерфейс, в котором может быть только один абстрактный метод, используется для лямбда-выражений. При этом нет ограничений на остальные методы (дефолтные, статические). Исключение составляют методы класса Object, которые могут быть объявлены в интерфейсе как абстрактные, но так как все классы в Java являются наследниками Object и, соответственно, его реализаций, то такому методу будет присвоена реализация по умолчанию и он, вроде как, не абстрактный (объявляется для уточнения контракта, его нельзя реализовать как default метод)

Интерфейс-маркер – интерфейс, не имеющий ни одного абстрактного метода, имплементирующему классу дает только тип.

**Виды классов:**

*Внешние классы* – Обычный класс, который может содержать классы, описанные ниже. Внешние классы могут быть только public и package-private.

*Внутренние* (nested classes) – Классы, определённые внутри внешнего/обрамляющего класса. Подразделяются на static nested classes и non-static nested classes (включающие inner, local, anonymous). Любой внутренний класс создается для обслуживания внешнего класса (вспомогательный код для обрамляющего) и обеспечения дополнительного уровня инкапсуляции (скрывает переменные внутри себя, доступ к ним остается только у обрамляющего (private переменные доступны из внешнего класса) – при правильном модификаторе доступа). К побочным плюсам относятся – логическая группировка классов и создание более читаемого кода.

**

*Статический вложенный класс (static nested class) & Нестатический вложенный класс (Inner classes).*

|  |  |  |
| --- | --- | --- |
|  | *Статический вложенный класс (static nested class)* | *Нестатический вложенный класс (Inner classes)* |
| Создание объекта (главное отличие) | Не привязан к объекту обрамляющего класса. Объект static класса может существовать без создания объекта внешнего класса. | При создании объекта внутреннего класса, обязательно создание или указание обрамляющего класса, к которому будет привязан объект.  Каждый объект внутреннего класса хранит в себе ссылку на привязанный к нему объект обрамляющего класса (аналогично нестатическим методам, которые хранят вызвавший их объект - this). |
| Обращение к полям и методам | Накладывается ограничение на обращение из non-static методам внутреннего класса к non-static методу внешнего, т.к. С.В.К. не привязан к созданию объекта обрамляющего класса, обращение к non-static методу/полю обрамляющего класса не дает гарантии, что объект обрамляющего класса вообще существует. Остальные взаимодействия между классами работают как обычно. | Взаимодействия между объектами и полями класса действуют по обычным правилам. |
| Обычные взаимодействия между членами класса:  Из static в non-static – нельзя без объекта  Из non-static в static - можно  Из static в static - можно  Из non-static в non-static – можно, т.к. во втором методе берется неявный параметр this, который является объектом, на котором вызвали метод. | | |
| Объявление полей и static методов | Разрешено объявление любых полей и методов. | До Java 16 нельзя было создавать static поля и методы, только static final поля, сейчас ограничений нет. |
| Посыл | Доп. функционал для класса - отражает Агрегацию. | Доп. функционал для объекта класса – отражает Композицию. |
| Синтаксис создания | Объект создается через обращение к обрамляющему классу.  A.B b = new A.B(); | Создается так же через обрамляющий класс, но подразумевает привязку к объекту обрамляющего класса путем создания нового объекта или привязки к существующему  A.B b = new A().new B();  Или  A a = new A(); A.B b = a.new B(); |
| Наследование | Может наследоваться и быть наследуемым любым классом, включая любой внутренний класс, если соответствует модификатор доступа. | Может наследовать любой класс.  От него могут наследоваться:  1.Другой внутренний класс, объявленный в том же самом внешнем классе или в его наследнике.  2. Обычный класс.  3. Внутренний класс, объявленный в другом классе.  Во 2 и 3 случае в конструктор требуется явная передача объекта внешнего класса и вызов у него метода super(), т.к. он должен быть связан с объектом обрамляющего класса.  public class B { public class C {} //внутренний класс }  public class Main {  class A extends B.C {  A(B b) { //ручная передача объекта обрамляющего класса  b.super(); //вызов конструктора обрамляющего класса  }  } } |

*Локальный класс (Method local inner class)* – Класс, объявленный внутри обрамляющего класса и некоторого блока кода (метода, просто блока кода, конструктора, цикла, if). При этом этот класс не принадлежит обрамляющему классу, а принадлежит блоку кода, в котором он был объявлен. Рассматривается как локальная переменная этого блока кода (в отношении доступа и существования).

1) Объект класса может быть создан только в блоке, в котором объявлен класс.

2) Может использовать все переменные и методы обрамляющего класса (независимо от модификатора доступа).

3) Может использовать аргументы и переменные блока кода, в котором объявлен, если они final или effective-final – переменная, которая не изменяет своего значения после объявления. Соответственно эти переменные нельзя изменять внутри этого класса.

До Java 8 можно было использовать только final аргументы или методы.

void foo(final int a) //final аргумент  
{  
 final int b = 0;// final переменная  
 class A {  
 void aa() {  
 System.*out*.println(a);// использование аргумента метода  
 System.*out*.println(b);// использование переменной метода  
 }  
 }  
}

После Java 8

void foo(int a) // effective-final аргумент  
{  
 int b = 0;// effective-final переменная  
 int c = 11;  
 c = 12; //изменили переменную  
 class A {  
 void aa() {  
 System.*out*.println(a);  
 System.*out*.println(b);  
 System.*out*.println(c);//ошибка, переменная с не является effective-final  
 }  
 }  
}

4) Если объявлен в static методе, то не может напрямую вызывать нестатические члены класса, так как объект обрамляющего класса может просто отсутствовать, поэтому для обращения к non-static полям класса нужно обратиться к уже существующему объекту или создать свой.

public class Main {  
 void foo() {  
 }  
  
 public static void main(String[] args) {  
 Main main = new Main(); //явно создаем объект Main  
 class A { //создан в static методе  
 void doA() {  
 main.foo(); //обращение к non-static методу Main  
 }  
 }  
 }  
}

При объявлении в non-static методе обрамляющего класса гарантируется, что его объект уже создан и можно обратиться к любому non-static методу (будет использован параметр this)

public class Main {  
 void foo() {  
 }  
 void fooo() {  
 class A { //создали класс A в нестатической среде  
 void asd() {//вызвали метод foo() класса Main  
 foo();//неявно вызвается Main.this.foo()  
 }  
 }  
 }  
}

5) Не могли иметь static переменных и методов до Java 16 (сейчас доступно объявление всех видов переменных)

6) Может иметь только package-private доступ к классу

При этом поля локального класса могут иметь модификаторы доступа – непонятно для чего, т.к. независимо от модификатора доступа этот класс доступен только в объявленном блоке кода\*\*

\*Локальный и вложенный класс может быть, как абстрактным, так и объявленным интерфейсом\* - с Java 16.

***Использование:***

Если нужно создать экземпляр класса более одного раза

Если есть необходимость в конструкторе

Основное применение локальные классы находят в тех случаях, когда необходимо написать класс, который будет использоваться внутри одного метода. Создание локального класса – способ не загромождать пространство имен.

*Анонимные классы –* Частный случай локального класса, обладает теми же свойствами, что и локальный класс (в теле можно объявлять переменные, методы, имеется доступ ко всем членам обрамляющего класса). Может быть применен как к обычному классу, так и к интерфейсу или абстрактному классу. Нет возможности задать какой-либо модификатор доступа для этого класса, так как объявление класса скрыто от пользователя.

Отличие от локального:

1) Не задает имя класса (не задается явно, компилятор сгенерирует имя типа Test.Main$1)

2) Нельзя явно инициализировать собственный конструктор, будет использован конструктор базового класса, какой именно, будет зависеть от переданных параметров.

Под капотом:

1) Создается класс, с некоторым именем, которое недоступно программисту, который будет реализовывать интерфейс, абстрактный класс или обычный класс.

2) В теле класса реализуются все методы интерфейса или абстрактные методы класса либо же методы, которые необходимо переопределить.

3) Создается объект этого класса.

Применение:

1) В случае, если необходим локальный класс для одноразового использования (класс, с переопределенным методом, который будет использован 1 раз) – чаще всего используются для реализации абстрактного класса или интерфейса.

2) Позволяет избежать написания большого кол-ва кода руками и не захламлять пространство классовых имен (не нужно явно создавать класс, придумывать ему имя и наследовать класс или интерфейс)

Анонимный класс может быть создан на основе любого класса, механика такая же. См пример.

Синтаксис:

public class Main {  
 private int number;  
 private static int *number1*;  
 private void foo() {}  
 public static void main(String[] args) {  
 Extendable ext = new Extendable(1, 1) { //создание анонимного класса на основе класса Extendable, выбрали один из его конструкторов посредством параметров.   
 int e;  
 static int *ee*;  
 static void boo() {  
 }  
 void boos() {  
 } //можно определять любой член класса, как в обычном классе, кроме конструктора, он берется от родителя  
 @Override  
 void voo() {// определяем собственную реализацию для метода родителя  
 *ee* = Main.*number1*;  
 this.e = new Main().number; //позволяет использовать любой член обрамляющего класса, независимо от модификатора  
 new Main().foo();  
 }  
 }; //{} - подразумевает тело класса, (); - говорит о создании объекта  
 }  
}

class Extendable{  
 Extendable(int a, int b){}  
 Extendable(int a){}  
 void voo(){}  
 static void voos(){}  
}

Модификаторы доступа для разных видов класса:

|  |  |
| --- | --- |
| Внешний | Package-private или public |
| Статический внутренний | Любой |
| Внутренний | Любой |
| Локальный | Всегда package-private |
| Анонимный | Нельзя задать модификатор доступа (всегда package-private) |

Основная задумка заключается в жизненном цикле объектов. Несмотря на локальный характер локальных и анонимных классов мы можем использовать их экземпляры за пределами тех блоков, в которых они были объявлены (через присвоение объекта какому-нибудь полю или через возврат из метода). Стоит помнить, что жизненный цикл таких объектов может продолжаться даже после завершения их "родного" блока, ведь объект не рассматривается целью для сборщика мусора до тех пор, пока на него есть активные (не перекрёстные) ссылки. Таким образом, если бы мы оставили всё, как есть, то могла бы возникнуть ситуация, что мы обращаемся внутри локального класса к переменной, которой уже не существует, так как объект всё ещё живой, а блок уже своё отработал. Именно поэтому и создаётся копия локальных переменных. Это основной лейтмотив, который стоит уяснить, как мне кажется. Дополнительно стоит уточнить процесс создания копии локальных переменных. При создании экземпляра анонимного или локального класса, который ссылается на внешние переменные, эти переменные фактически "копируются" в объект. Причина этому – гарантировать, что даже если исходная переменная во внешнем коде будет изменена, значение внутри объекта останется стабильным и неизменным. По этой же причине требуется, чтобы локальные переменные были объявлены как final или, по крайней мере, фактически были финальными (то есть не изменялись после инициализации). Это обеспечивает стабильность и предсказуемость поведения, так как код внутри анонимного или локального класса всегда будет работать с тем же значением, которое было в момент его создания, независимо от того, что происходит во внешнем коде.

Пример кода, в котором объект существует после завершения метода, и при этом переменная должна была быть затерта, если б не копировалась в класс:

class Main {  
 public static void main(String[] args) {  
 Runnable run = new Y().f();  
 run.run();// в момент вызова метода переменной p уже не должно существовать, так как мы вышли из метода f()

}  
}  
class Y {  
 public Runnable f() {  
 int p = 1;  
 return new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println(p);  
 }  
 };  
 }  
}

Компиляция внутренних классов

Локальные/анонимные классы компилируются в обычные внутренние классы. Они получают копии аргументов класса и всех используемых effective-final переменных в виде собственных полей.

**Final**

|  |  |
| --- | --- |
| Переменная | Нельзя присваивать новое значение переменной, но можно изменять содержимое, если это ссылка |
| Метод | Виртуальный: Нельзя переопределять метод, private методы неявно final  Static: запрещает перекрытие в наследниках |
| Аргумент метода | Нельзя присваивать новое значение аргументу метода  void foo(final String a) { } |
| Класс | Нельзя наследоваться от него |
| Интерфейс | Не используется, т.к подразумевается, что от интерфейса будут наследоваться. |
| Абстрактный класс | Не используется, т.к подразумевается, что от класса будут наследоваться. |

***Конструкторы классов***

Специальный метод каждого класса. Вызывается оператором new (обычный метод вызывается через объект). Используется как шаблон для инициализации полей объекта.

Имя – совпадает с именем класса. Имеет параметры доступа. Не имеет возвращаемого типа (даже void) – возвращают объект класса. Могут принимать в себя параметры и имеют тело. Могут быть перегружены, как и методы.

Конструктор класса создается по умолчанию, даже если явно не прописан. Конструкторы не наследуются, но в дочернем классе требуют обращения к конструктору класса родителя через super(). Конструкторы не наследуются по причине того, что у дочернего класса может быть больше полей, которые нужно инициализировать, вызов конструктора родителя требуется из-за возможности наследования предком private полей.

**Блоки инициализации:**

Статический блок инициализации – Исполняется единожды, при загрузке класса, используется для инициализации статических переменных, если для их инициализации необходима какая-либо логика отличная от простого присваивания.

Нестатический блок инициализации – Исполняется каждый раз при создании объекта до вызова конструктора этого объекта, так же используются для реализации более сложной логики, отличной от простого присваивания.