**Внутренние классы**

Нестатические вложенные классы называют также внутренними классами. Внутренний класс имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них. Внутренние классы создаются внутри окружающего класса :

// внешний класс

class Outer {

int x1 = 0;

int x2 = 0;

void summa(final int x1, final int x2) {

this.x1 = x1;

this.x2 = x2;

Inner inner = new Inner();

inner.display();

}

// внутренний класс

class Inner {

void display() {

System.out.println ("summa = " + String.valueOf(x1 + x2));

}

}

}

...

class MainActivity {

Outer outer = new Outer();

outer.summa(12, 11);

}

**Статические внутренние классы**

Статические внутренние классы декларируются внутри основного класса и обозначаются ключевым словом **static**. Они не имеют доступа к членам внешнего класса за исключением статических.

Статический внутренний класс может содержать статические поля, методы и классы, в отличие от других типов внутренних классов. Пример :

class OuterClass

{

private int outerField;

static int staticOuterField;

public OuterClass() {}

static class InnerClass

{

int getOuterField()

{

return OuterClass.this.outerField; // Эта линия кода вызывает ошибку при компиляции

}

int getStaticOuterField()

{

return OuterClass.staticOuterField; // Эта линия кода синтаксически корректна

}

}

}

**Локальные классы**

Локальные классы объявляются внутри методов основного класса и могут быть использованы только внутри этих методов. Они имеют доступ к членам внешнего класса, а также как к локальным переменным, так и к параметрам метода при одном условии - переменные и параметры используемые локальным классом должны быть задекларированы *final*. **Локальные классы** не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант). Пример :

class OuterClass

{

public OuterClass(){}

private int outerField;

InnerClass inner; // Эта линия кода вызывает ошибку при компиляции

void methodWithLocalClass (final int parameter)

{

InnerClass innerInsideMehod; // Эта линия кода синтаксически корректна

int notFinal = 0;

class InnerClass

{

int getOuterField()

{

return OuterClass.this.outerField; // Эта линия кода синтаксически корректна

}

notFinal++; // Эта линия кода вызывает ошибку при компиляции

int getParameter()

{

return parameter; // Эта линия кода синтаксически корректна

}

};

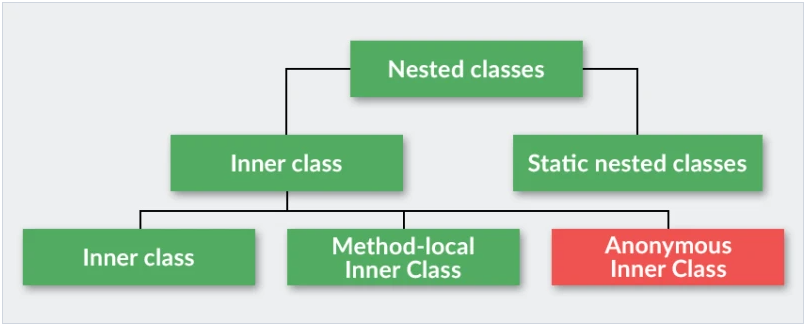
}

};

Анонимные классы в Java

В документации Oracle приведена хорошая [рекомендация](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html#accessing):

**«Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования»**.



Как и локальные классы, о которых мы говорили в прошлой лекции, анонимные —подвид внутренних классов. У них также есть несколько сходств и отличий между собой. Но для начала давай разберемся: а почему они, собственно, называются «анонимными»? Для этого рассмотрим простой пример. Представь, что у нас есть основная программа, которая постоянно работает и что-то делает. Мы хотим создать для этой программы систему мониторинга из нескольких модулей. Один модуль будет отслеживать общие показатели работы и вести лог, второй — фиксировать и регистрировать ошибки в журнале ошибок, третий — отслеживать подозрительную активность: например, попытки несанкционированного доступа и прочие связанные с безопасностью вещи. Поскольку все три модуля должны, по сути, просто стартовать в начале программы и работать в фоновом режиме, будет хорошей идеей создать для них общий интерфейс:

**public** **interface** MonitoringSystem {

**public** **void** startMonitoring();

}

Его будут имплементировать 3 конкретных класса:

**public** **class** GeneralIndicatorsMonitoringModule **implements** MonitoringSystem {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг общих показателей стартовал!");

}

}

**public** **class** ErrorMonitoringModule **implements** MonitoringSystem {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг отслеживания ошибок стартовал!");

}

}

**public** **class** SecurityModule **implements** MonitoringSystem {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг безопасности стартовал!");

}

}

Казалось бы, все в порядке. У нас есть довольно внятная система из нескольких модулей. У каждого из них есть собственное поведение. Если нам понадобятся новые модули, мы сможем их добавить, ведь у нас есть интерфейс, который достаточно легко имплементировать. Но давай подумаем о том, как будет работать наша система мониторинга. По сути, мы должны просто создать 3 объекта — GeneralIndicatorsMonitoringModule, ErrorMonitoringModule, SecurityModule — и вызвать метод startMonitoring() у каждого из них. То есть, все, что нужно сделать — создать 3 объекта и вызвать у них 1 метод.

**public** **class** Main {

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

GeneralIndicatorsMonitoringModule generalModule = **new** GeneralIndicatorsMonitoringModule();

ErrorMonitoringModule errorModule = **new** ErrorMonitoringModule();

SecurityModule securityModule = **new** SecurityModule();

generalModule.startMonitoring();

errorModule.startMonitoring();

securityModule.startMonitoring();

}

}

Вывод в консоль:

*Мониторинг общих показателей стартовал!*

*Мониторинг отслеживания ошибок стартовал!*

*Мониторинг безопасности стартовал!*

И для такой небольшой работы мы написали целую систему: 3 класса и один интерфейс! И все это — ради 6 строк кода. С другой стороны, какие у нас варианты? Да, не очень здорово, что мы понаписали таких вот «одноразовых» классов. Но как мы можем это исправить? Здесь нам и приходят на помощь **анонимные внутренние классы**! Вот как они выглядят в нашем случае:

**public** **class** Main {

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

MonitoringSystem generalModule = **new** MonitoringSystem() {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг общих показателей стартовал!");

}

};

MonitoringSystem errorModule = **new** MonitoringSystem() {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг отслеживания ошибок стартовал!");

}

};

MonitoringSystem securityModule = **new** MonitoringSystem() {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг безопасности стартовал!");

}

};

generalModule.startMonitoring();

errorModule.startMonitoring();

securityModule.startMonitoring();

}

}

Давай разбираться, что тут происходит! Выглядит так, как будто мы создаем объект интерфейса:

MonitoringSystem generalModule = **new** MonitoringSystem() {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг общих показателей стартовал!");

}

};

Но ведь мы давно знаем, что создавать объекты интерфейсов нельзя! Так и есть, нельзя. На самом деле мы этого и не делаем. В тот момент, когда мы пишем:

MonitoringSystem generalModule = **new** MonitoringSystem() {

};

внутри Java-машины происходит следующее:

1. Создается безымянный Java-класс, реализующий интерфейс MonitoringSystem.
2. Компилятор, увидев такой класс, требует от тебя реализовать все методы интерфейса MonitoringSystem (мы это и сделали 3 раза).
3. Создается один объект этого класса. Обрати внимание на код:

MonitoringSystem generalModule = **new** MonitoringSystem() {

};

В конце стоит точка с запятой! Она стоит там не просто так. Мы одновременно объявляем класс (посредством фигурных скобок) и создаем его объект с помощью (); Каждый из наших трех объектов переопределил метод startMonitoring() по-своему. В конце мы просто вызываем этот метод у каждого из них:

generalModule.startMonitoring();

errorModule.startMonitoring();

securityModule.startMonitoring();

Вывод в консоль:

*Мониторинг общих показателей стартовал!*

*Мониторинг отслеживания ошибок стартовал!*

*Мониторинг безопасности стартовал!*

Вот и все! Мы выполнили свою задачу: создали три объекта MonitoringSystem, переопределили его тремя разными способами и вызвали трижды. Все три модуля успешно запущены и работают. При этом структура нашей программы стала намного проще! Ведь классы GeneralIndicatorsMonitoringModule, ErrorMonitoringModule, SecurityModule теперь вообще можно удалить из программы! Они нам просто не нужны — мы прекрасно справились и без них. Если каждому из наших анонимных классов-модулей понадобится какое-то отличающееся поведение, свои специфические методы, которых нет у других, мы легко можем дописать их:

MonitoringSystem generalModule = **new** MonitoringSystem() {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг общих показателей стартовал!");

}

**public** **void** someSpecificMethod() {

System.out.println("Специфический метод только для первого модуля");

}

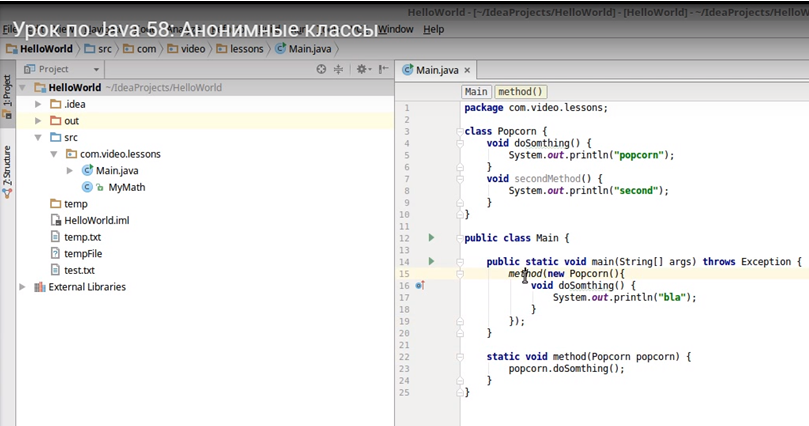
};

Так же мы можем переопределять и поведение любого готового класса используя Анонимный клас:



Хотя чаще всего это делается для определения интерфейсов.

Еще Анонимный класс может использоваться как аргумент:



Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным:

**public** **class** Main {

**private** **static** **int** currentErrorsCount = 23;

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

MonitoringSystem errorModule = **new** MonitoringSystem() {

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг отслеживания ошибок стартовал!");

}

**public** **int** getCurrentErrorsCount() {

**return** currentErrorsCount;

}

};

}

}

Есть у них кое-что общее и с локальными классами: они видны только внутри того метода, в котором определены. В примере выше, любые попытки обратиться к объекту errorModule за пределами метода main() будут неудачными. И еще одно важное ограничение, которое досталось анонимным классам от их «предков» — внутренних классов: **анонимный класс не может содержать статические переменные и методы**. Если мы попробуем сделать метод getCurrentErrorsCount() из примера выше статическим, компилятор выбросит ошибку:

//ошибка! Inner classes cannot have static declarations

**public** **static** **int** getCurrentErrorsCount() {

**return** currentErrorsCount;

}

Тот же результат мы получим, если попробуем объявить статическую переменную:

MonitoringSystem errorModule = **new** MonitoringSystem() {

//ошибка! Inner classes cannot have static declarations!

**static** **int** staticInt = 10;

@Override

**public** **void** startMonitoring() {

System.out.println("Мониторинг отслеживания ошибок стартовал!");

}

};

Напоследок могу порекомендовать тебе отличное видео по теме анонимных классов, где данная тема объясняется максимально просто и понятно :)

https://www.youtube.com/watch?v=ndnubpPzkNE&t=5s