ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

Программирование на Java (2022 - 2023)

Лекция 5.

Абстрактные классы и интерфейсы.

Вложенные интерфейсы и классы

к.т.н. Гринкруг Е.М. (email: egrinkrug@hse.ru)

Абстрактные классы и методы

- Если необходимо определить некоторую совокупность методов, которыми должны обладать все производные классы, **но не надо создавать объекты базового класса**, используют понятия абстрактного метода и абстрактного класса (*чтобы «навязать» потомкам частично реализованную функциональность…*).
- Абстрактный метод состоит только из декларации и не имеет тела метода, например: abstract void f (int x);
- Класс, содержащий хоть один абстрактный метод, обязан быть абстрактным классом, что декларируется соответственно: abstract class C { … };
- Класс, производный от абстрактного, обязан предоставить реализации всех абстрактных методов абстрактного класса, иначе его самого придется сделать абстрактным;
- Класс можно объявить абстрактным и без абстрактных методов.
- **Инстанциировать абстрактный класс нельзя** , что может быть полезно само по себе ...

Как это можно сделать иначе? Зачем это может быть нужно?

Интерфейсы

- ***Абстрактные классы могут иметь «частичную» реализацию***, наследуемую производными от них классами;
- ***Интерфейсы*** - следующий шаг абстракции – **изначально могли *определять* только абстрактные методы** (без реализации) **и статические константы:**
  - **до JDK 8 вообще ни один метод интерфейса не мог содержать реализацию**;
  - **с JDK 8 бывают методы интерфейса с реализацией, но об этом – чуть позже...**
- Интерфейсы декларируются с ключевым словом **interface** (вместо **class**), что означает (примерно): *«так должны выглядеть (и уметь делать) все классы, которые **реализуют** данный интерфейс»*;
- Суть понятия «интерфейс» – протокол взаимодействия между классами (без какого-либо намека на способ реализации);
- Интерфейс и полностью абстрактный класс отличаются тем, что можно наследовать только один абстрактный класс (наследуя *состояние*), но реализовывать можно много интерфейсов (не имея в них *состояний*).

- Абстрактный класс может определять статические поля и поля экземпляров, задавая тем самым некоторую структуру реализации для объектов производных классов («состояние» объектов);
- Интерфейсы тоже могут содержать поля, но они автоматически являются **public**, **static** и **final** (т.е. общими статическими константами);
- Для описания класса, реализующего набор интерфейсов, используется ключевое слово «implements»:

  class *MyClass* ***implements*** *MyInterface1, MyInterface2 { /* … */}*
- Все абстрактные методы, описанные в интерфейсе, автоматически являются **public** (у них можно опускать **public**); при реализации метода интерфейса в некотором классе метод должен быть объявлен **public;** в JDK8 появились и private-методы в интерфейсах… (об этом далее)…
  - ***компилятор запрещает понижать уровень доступа при наследовании и реализации интерфейсов (повышать можно)*;**

Отделение интерфейса от реализации

- Интерфейс позволяет иметь его полностью независимые реализации (тогда как наследование базового класса подразумевает и [частичное] наследование реализации);
- Как и «в реальной жизни», если взаимодействие через интерфейс не получается (сходная функциональность реализована без использования данного интерфейса), можно использовать интерфейс-адаптер (ср., например, вилки питания в Европе и США и их адаптеры);

Свойства интерфейсов

- Интерфейсы (как и абстрактные классы) нельзя инстанциировать (то есть нельзя создать объект, не зная его реализации…);
- Интерфейс (как и класс) определяет ссылочный тип данных, который может использоваться для декларации переменных:

```
MyInterface1  myObject;
```

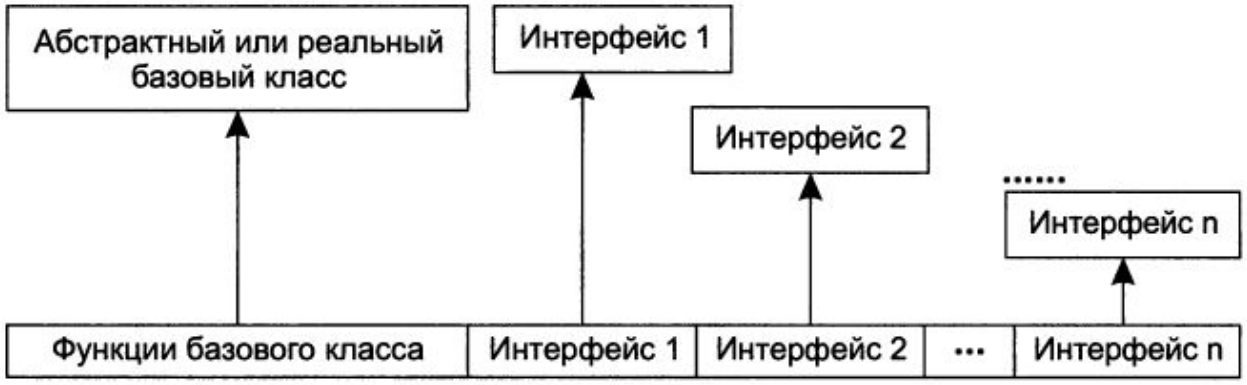
- При этом переменная «интефейсного типа» должна ссылаться на объект класса, который реализует этот интерфейс;
- Оператор **instanceof** работает для классов и для интерфейсов;
- Интерфейсы можно наследовать в интерфейсе (причем более одного):

```
public interface Moveable { void move (double x, double y); }
public interface Nameable { String getName(); }
public interface Powered extends Moveable, Nameable {
  double milesPerLitre();  double SPEED_LIMIT = 95;
}
```

- Интерфейсы могут определять только константы (не иметь методов), однако это – не есть хороший стиль программирования;
- Интерфейсы в Java играют роль средства, сходного с множественным наследованием (**которого в Java нет, хотя с Java 8 появились некоторые слегка «похожие» возможности – о них чуть позже...**).

«Множественное наследование» и Java

- Один и тот же объект часто должен выступать в разных ипостасях: «мой объект –***и швец, и жнец, и на дуде игрец***» (то есть, объект может рассматриваться как объект разных типов одновременно);
- В Java только один из типов может «диктовать» реализацию, а остальные типы обеспечиваются реализацией интерфейсов (и к этим типам тоже можно производить *восходящее преобразование*...):



- Класс, реализующий интерфейс, должен предоставить реализации всех абстрактных методов интерфейса;
- Если два разных интерфейса содержат совпадающие спецификации методов, реализация метода в классе, реализующем оба интерфейса, предоставляется только одна;
- Одновременные переопределение, реализация и перегрузка методов затрудняют чтение кода, лучше этого избегать;
- Главными причинами введения интерфейсов являются:
  - Отделение спецификации от реализации («что» от «как»);
  - Возможность восходящего приведения к нескольким типам;
  - Запрет создания экземпляров типов клиенту (как и в абстрактных классах).

«При прочих равных» следует отдавать предпочтение интерфейсам, а не абстрактным классам, - в тех случаях, где нужна большая гибкость

Интерфейсы как средство адаптации

- Одним из важнейших причин использования интерфейсов является возможность их разных реализаций.
- В простейших случаях, например, имеется метод, который принимает параметром объект интерфейсного типа, позволяя вам как угодно его реализовывать и использовать при вызове метода.
  - Например, интерфейс Readable был придуман специально для работы с конструктором Scanner(Readable) (см.);
  - Таким образом Scanner способен работать с большим разнообразием типов (таких, кто сразу сделан или будет сделан Readable);
- Обычно легко бывает устроить адаптер, наследуя имеющийся класс и добавляя к производному классу реализацию нужного интерфейса (иногда такое примешивание одного к другому называют “mix-in”). В этом преимущество (гибкость) интерфейсов по сравнению с классами.

Поля в интерфейсах

- Поля, помещаемые в интерфейсе, автоматически являются **static** и **final**, поэтому они годятся для создания именованных групп констант;
- До Java SE5 так имитировали перечисляемый тип **enum**, например:

```
public interface Months {
    int
            JANUARY = 1, FEBRUARY = 2, MARCH = 3,
            APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
            AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
            NOVEMBER = 11, DECEMBER = 12;
} ///·~
```

- В Java SE5 появилось ключевое слово **enum** (и надобность в использовании для этого полей в интерфейсах отпала, но примеры такого использования интерфейсов могут встретиться – в том числе в исходных файлах старых библиотек);

- Инициализация полей интерфейсов происходит аналогично инициализации статических полей классов – при первом использовании *(придумайте, как это протестировать?).* Поля интерфейсов должны быть явно инициализированы.

Статические методы в интерфейсах - JDK 8(+)

- Статические методы в интерфейсах разрешены, начиная с JDK 8;
  - **принципиальных причин для их запрета и не было: просто хотелось иметь в интерфейсах только абстрактные спецификации, без реализаций чего-либо...**
- Из-за этого запрета приходилось иметь «сопутствующие» утилитные классы (наборы соответствующих статических методов):
  - **для интерфейса Collection - класс Collections;**
  - **для интерфейса Path – класс Paths, …**
- В таких утилитных классах обычно есть лишь несколько фабричных методов, которые теперь можно задать в самих интерфейсах, наряду с другими статическими операциями с объектами интерфейсного типа.
- Едва ли из JDK будут убирать такие утилитные классы (*почему?*), но при внедрении новых интерфейсов уже нет нужды в изготовлении соответствующих им утилитных классов (методы могут быть прямо в них)…

default методы в интерфейсах - JDK 8(+)

- Начиная с JDK 8, можно задавать реализации и ***нестатических*** методов интерфейса, снабжая их ключевым словом default.
- Главная цель - развитие интерфейсов без влияния на имеющийся код.
  - **При добавлении нового метода к интерфейсу ранее надо было бы переделать все его существующие реализации. default-метод из новой версии интерфейса подставится в уже имеющиеся реализации интерфейса автоматически.**
- Кроме того, default-методы дают экономию, избавляя от необходимости реализации «ненужных» классов и/или методов:
  - **не нужны классы AbstractBaseImplementation и/или классы-адаптеры,**
  - **в immutable-коллекциях не нужны затычки методов remove() и т.п.**
- Для удобства реализации default-методов в интерфейсах JDK 9 стало можно реализовывать private-методы (*почему?*)
- **Интерфейсы по-прежнему не имеют инстанс-полей** (не определяют состояния объектов); они определяют «что» делается, а не «как» ...

Множественное наследование default-методов

- Как теперь быть, если в разных реализуемых интерфейсах и суперклассах есть одинаковые методы?
  - В языках с множественным наследованием есть правила разрешения конфликтов...
- default-методы – НЕ ЕСТЬ аналог множественного наследования:
  - Реализация интерфейсов не определяет состояние (не наследуются инстанс-поля).
- В Java 8(+) правила простые:
  - **Класс всегда выигрывает (если там такой метод есть, default-методы игнорируются; это обеспечивает обратную совместимость с JDK < 8);**
  - **Если default-методы переопределяются при наследовании самих интерфейсов, работает default-метод наследующего интерфейса;**
  - **Можно явно указать нужный: <интерфейс>.super.<default-метод>**
    - **Тут super – отсылка к реализуемому интерфейсу (новый контекст пользования super).**
- *Вопрос: есть ли смысл делать default-методы в интерфейсе, совпадающие с методами класса* java.lang.Object*?*

Вложенные интерфейсы

- Интерфейсы могут быть вложены в другие интерфейсы и классы. В этом случае они являются вложенными (nested) *(классы тоже могут быть вложенными, но об этом – чуть позже)*;
- При вложении интерфейса в класс *(см. примеры)* вложенный интерфейс может быть не только **public** или package-private (как обычно), но и **protected** или **private**;
  - Private nested interface – не может упоминаться вне класса его декларации (к нему нельзя сделать **upcast** из-вне класса его декларации); *зачем он может быть нужен?*
  - Единственный способ использования возвращаемого из метода охватывающего класса значения приватного интерфейсного типа – это использование его в методе этого же класса *(кто понял эту фразу?* ☺)

При вложении в интерфейс вложенный интерфейс может быть только public *(почему? проверьте, как обстоит дело с другими модификаторами доступа…)*

Интерфейсы и фабрики

- Интерфейс – это «переходник» к нескольким альтернативным реализациям;
- Типичным приемом получения объектов, соответствующих интерфейсу, является прием «фабрика» (factory pattern):
  - Вместо того, чтобы вызывать конструктор нужного класса непосредственно, вызывается метод объекта-фабрики (производящей нужные объекты);
  - При этом становится просто заменять разные реализации интерфейса;
  - Фабрики часто применяются при создании библиотек, при реализации параметризуемого создания объектов, и т.п.

    (кто пояснит смысл понятия “design pattern”?)

- Можно все программировать в интерфейсах, создавая инстансы фабриками.
- Интерфейс и его реализация с фабрикой могут заменить (почти) любой класс, но ценой дополнительного уровня абстракции и издержек времени. Поэтому, не надо увлекаться ими до абсурда.

Интерфейсы в старых и новых JDK

Итак, интерфейсы могут содержать следующие элементы:

- **В Java SE 7 и ранее:**
  - Константные «переменные» (это звучит дико ☺) - поля интерфейса;
  - Абстрактные методы;
- **В Java SE 8:**
  - Добавлены реализации статических методов;
    (зачем?)
  - Добавлены реализации **default**–методов;
    (зачем?)
- **В Java SE 9:**
  - Добавлены приватные статические и нестатические методы с реализацией в самом этом интерфейсе;
    (зачем?)
- Кроме того, начиная с JDK 1.1, интерфейсы могут содержать также вложенные интерфейсы и классы...

Важные «странные» интерфейсы Java

- Основной пакет Java – java.lang, в котором сосредоточены самые важные понятия языка. С самой первой версии там есть несколько интерфейсов, и среди них java.lang.Cloneable (см.):

```
package java.lang;
public interface Cloneable { }
```

(Интерфейсы, в которых нет членов, называют маркер-интерфейсами (tagging); их наличие позволяет просто «пометить» классы, что – не есть хорошо: интерфейсы нужны не для этого, и пометить классы можно иначе (например, - аннотациями). Однако, такие маркер-интерфейсы существуют и широко используются в работающих программах)

- Рассмотрим класс java.lang.Object – корневой для всех классов в java; Там есть метод ***(постепенно мы рассмотрим их все)***:

```
protected native Object clone() throws CloneNotSupportedException
```

(ключевое слово native означает, что метод реализован не средствами языка java, а средствами реализации java-машины)

- **Почему нельзя вызывать protected clone()-метод класса Object из любого класса, если любой класс – подкласс класса Object?**

```
public class Test implements Cloneable {
    public static void main(String[ ] args) {
        Object o = new Object();
        Object myCopy = o.clone();          // это не компилируется (почему?)...
    }
    Object makeCopy( ) {
        try{
                return super.clone();    // это компилируется (почему?)...
        } catch (Exception ex) {
            // that cannot be happened… (explain - why?)
        }
    }
}
```

- Модификатор protected означает, что член класса виден только классам из этого же пакета и наследникам, ***причем если наследник находится в другом пакете, он не имеет доступа к protected-члену через ссылку на объект не своего типа.***
- *Кто кому тут наследник? Что и почему мы не можем или можем выше сделать?*

- ***Только сам класс может клонировать свои объекты***. Чтобы клонировались другие (чужие) объекты, нужен их **public** метод **clone()**;
- Класс Object мог бы скопировать все поля, но
  - копирование значений полей ссылочных типов – просто копирует сами ссылки...(shallow copy – «поверхностное» копирование);
  - Иногда достаточно такого клонирования (объект может быть immutable, и т. п.), но иногда – надо делать «глубокое» копирование (deep copy);
- Для каждого класса надо понимать:
  - Достаточно ли метода clone() с shallow copy?
  - Надо ли дополнять его до deep copy?
  - Следует ли вообще применять метод clone()?

- Есть разные точки зрения по поводу ответа на последний вопрос
  - См. Effective Java, Edition 3, Item 13 – override clone judiciously;
  - Сравните с использованием метода finalize() – (см. там же, Item 8).

- При положительном решении об использовании clone(), надо:
  - **Сделать класс как *implements Cloneable*;**
    - **Такие «пустые» интерфейсы называют «интерфейсами-маркерами» или tagged interfaces; единственный их смысл – в проверке объекта на принадлежность к их типу (например, if (obj instanceof Cloneable)…).**
    - **Определение интерфейса Cloneable не содержит методов (!);**
    - **Что же делает Cloneable, если там нет методов? Он определяет реализацию метода clone() в классе java.lang.Object:**
      - **Если класс implements Cloneable, метод делает просто копии всех полей объекта;**
      - **Иначе – выбрасывает CloneNotSupportedException.**
    - **Обработку этого исключения лучше не «затыкать», чтобы легче находить ошибки.**
  - **Переопределить в классе метод clone() с модификатором public**
    - **чтобы можно было клонировать объекты Cloneable-класса из-вне его самого…**

  Кстати: «в какую сторону» можно изменять права доступа при переопределении методов?

- Метод clone() иногда называют «псевдоконструктором» (как и метод readObject(), связанный с еще одним важным маркерным интерфейсом – java.io.Serializable): оба этих метода создают экземпляр объекта ***без вызова конструктора*** *(как это проверить для clone()?)*
- Таким образом, мы встречаемся тут с некоторой встроенной в JVM функциональностью, которая ***выходит за рамки языка***!
- Клонирование надо применять с осторожностью, - потому clone() и сделан protected в классе Object;
- Полезно рассматривать альтернативные варианты копирования с помощью copy-конструктора (и/или copy-фабрики);
- **Все массивы (любых типов элементов!) в Java имеют public (!) clone() метод и реализуют java.lang.Cloneable-интерфейс** (а также java.io.Serializable -интерфейс, о чем – позже, при разговоре о средствах ввода-вывода...).
- *Вопрос для размышления*: ***можно ли (если можно, то - как) отличить «клона», сделанного методом clone()) от «двойника» (сделанного copy-конструктором)?***

Вложенные классы

- **Java позволяет определять класс внутри другого класса. Класс, определенный внутри внешнего (охватывающего) класса, называется *вложенным классом (nested class*** – *терминология фирмы Sun)*
- **Вложенные классы распадаются на две категории:**
  - ***Статические*** **(static) вложенные классы;**
  - ***Нестатические*** **(non-static) вложенные классы, называемые также *внутренними (inner class) :***

```
class OuterClass {
        ...
        static class NestedClass { ... }
        class InnerClass { ... }
        …
}
```

- Любой вложенный класс является членом охватывающего класса и – как таковой – может быть объявлен **private, public, protected** или быть ***package private***. *(Какими могут быть классы, которые не являются вложенными?)*
- **Статические** вложенные классы **не имеют прямого доступа к** другим – ***нестатическим -*** членам охватывающего класса, но имеют – к **статическим, включая приватные!** *(тут у вас могут возникнуть вопросы...- они есть?)*
- Нестатические вложенные классы ***(inner-классы)*** *имеют непосредственный доступ и к другим членам охватывающего класса,* ***включая приватные****.*
  (тут у вас тоже могут возникнуть вопросы...- они есть?)
- Причин для использования вложенных классов несколько, в том числе:
  - Логическая группировка классов (например, нужных не всем);
  - Усиление средств инкапсуляции;
  - Улучшение структуризации кода (читаемость, и пр.).

- **Работу вложенных классов обеспечивает компилятор; для виртуальной машины нет отличия от внешних классов. Для обозначения вложенного класса в виртуальной машине используется символ $ (разделяет имена внешних и внутренних классов).**

Статические вложенные классы

- Как и другие статические члены класса (методы и поля), статические вложенные классы ассоциированы **с охватывающим классом**;
- Как и статические методы, они не могут непосредственно ссылаться на нестатические поля или методы охватывающего класса (без получения ссылки на объект); но могут обращаться к статическим членам охватывающего класса (полям, методам и *(всяким!)* вложенным классам), ***включая приватные***. В остальном, с позиций поведения, такие классы ничем не отличаются от top-level классов.
- Обращение к статическим вложенным классам (из-вне охватывающего их) происходит с использованием имени (имен) охватывающего, например:

```
MyOuterClass.MyStaticNestedClass myObject = new
    MyOuterClass.MyStaticNestedClass();
```

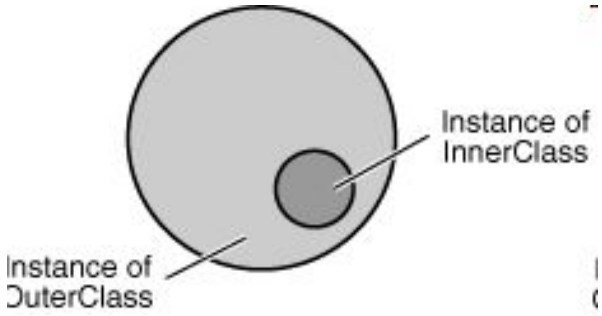
  - **в сущности, имя охватывающего выступает в роли «имени пакета».**

Внутренние классы

- Как и в случае полей и методов, нестатические вложенные классы (то есть – внутренние, ***inner***) ассоциируются **с экземпляром объекта охватывающего класса;**
- При этом имеется прямой доступ к полям и методам этого охватывающего объекта (***включая приватные***).
- До JDK16 во внутреннем классе нельзя было определять статические члены, так как он связан с экземпляром охватывающего класса. ***(Но во внутреннем классе можно было определять статические константы времени компиляции).*** Начиная с JDK16(+) это ограничение снято (это сделано в связи с внедрением Record(s), разновидности классов, о которых мы поговорим позднее).
- Объект (инстанс) внутреннего класса может существовать только внутри объекта охватывающего класса, имея прямой доступ ко всем полям и методам охватывающего объекта.
- Чтобы инстанциировать внутренний класс, надо сперва иметь объект охватывающего класса и просить его создать инстанс внутреннего класса:

  OuterClass.InnerClass innerObject = outerObject.new InnerClass();

- При создании объекта внутреннего класса в его конструктор автоматически (неявно, первым параметром) передается ссылка на объект охватывающего класса; эта ссылка всегда есть в распоряжении объекта внутреннего класса.
- **Даже если в конструкторе объекта внутреннего класса нет параметров, компилятор сам добавит параметр, сделав передачу ссылки на объект охватывающего класса в конструктор.**
- Во внутреннем классе эту ссылку на объект внешнего класса можно получить, записав **<имя внешнего класса>.this**;



При создании объекта внутреннего класса указывается не имя внешнего класса, а его объект, например:

Inner someInnerObject = new Outer().new Inner();

Внутренние классы и реализации интерфейсов

- В сущности, объект внутреннего класса является «оберткой» (wrapper’ ом) своего объекта охватывающего класса, ко всем «внутренностям» которого этот wrapper имеет прямой доступ (так как всегда имеет ссылку на него).

- Это удобно использовать для реализации интерфейса, которую можно абсолютно скрыть от окружающего мира. Приватный внутренний класс позволяет полностью скрыть все детали реализации.

- Охватывающий класс при этом может играть роль:
  - **Хранилища приватных данных;**
  - **Фабрики реализаций интерфейсов (использующих эти данные и реализованных с применением приватных внутренних классов).**
    (в условия полной изоляции реализации интерфейса компилятор имеет больше возможностей оптимизировать код реализации интерфейса)

Пример использования для реализации интерфейсов

```
public interface View2D { void draw(…); } // interface to paint 2D-image into some screen area…
public interface View3D { void draw(…); } // interface to paint 3D-image into some screen area…

public interface View2DProvider { View2D createView2D(); } // интерфейсы – фабрики...
public interface View3DProvider { View3D createView3D(); }

public class ObjectViewFactory implements View2DProvider, View3DProvider {
      // private data: shape, size, appearance, …
      private class View2DImpl implements View2D {
        //… implementation details…
        public void draw() { /* … */ }
      }
      private class View3DImpl implements View3D {
        // … implementation details…
        public void draw() { /* … */ }
      }
      public View2D createView2D(){ return this.new View2DImpl();} // фабрика для View2D
      public View3D createView3D(){ return this.new View3DImpl();} // фабрика для View3D
}
```

Использование в callback-механизмах

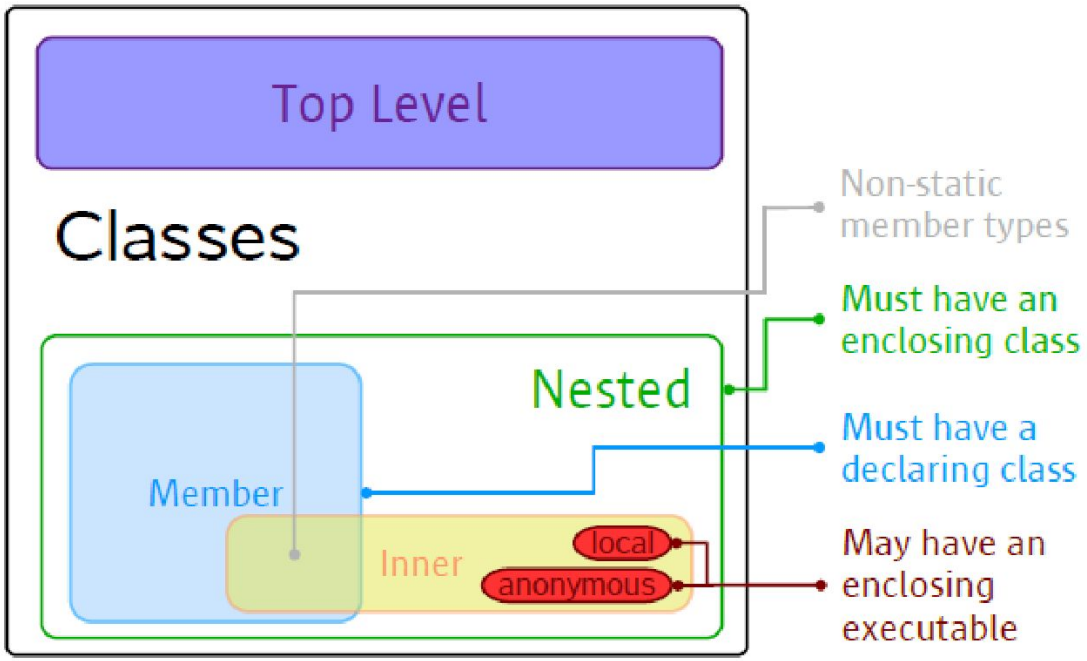
- Пример интерфейса для callback-механизма:

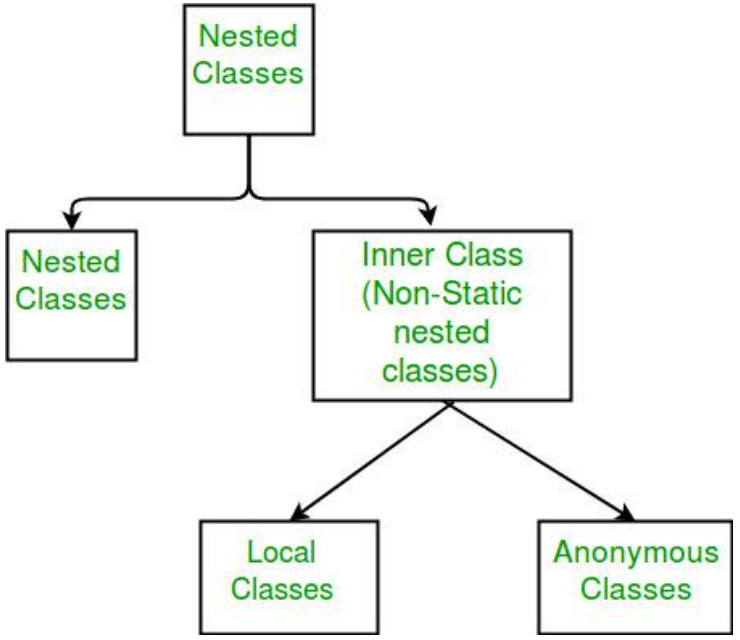
```
public interface
RandomNumberListener {
   void numberGenerated (int n);
}
```

```
public class OddNumbersLover { //odd - нечетный
        private Number myBestNumber = 1;
private void setBestNumber( int n){
                myBestNumber = n;
        }
        public RandomNumberListener
getRundomNumberListener() {
                        return new Listener();
                }
        private class Listener
          implements RandomNumberListener {
             public void numberGenerated (int  n) {
           if ((n % 2) != 0)
            setBestNumber(n);
                }
        }
   }
```

Кроме того, **имеются две специальные разновидности внутренних классов:**

- **локальные внутренние классы (local inner classes);**
- **анонимные внутренние классы (anonymous inner classes).**





Локальный внутренний класс

- Локальный внутренний класс может быть определен внутри метода, например:

```
public MyInterface getMyInterfaceReference (/* ... параметры метода...*/) {
   class MyInterfaceImpl implements MyInterface {
      private MyInterfaceImpl (/*... параметры конструктора... */) {
      }
      //…  прочие члены локального внутреннего класса
   }
   return new MyInterfaceImpl (/*…*/);
}
```

- Вложенный класс теперь не часть класса, а часть метода, и не может быть доступен вне его.
- Конфликт имен не происходит: имена классов генерируются (*см. - как!*).

Внутренний класс, локальный в блоке

- Область определения класса может быть сужена до блока, например:

```
public MyInterface getMyInterfaceReference (/* ... параметры метода...*/) {
    if ( isPermitted (…) )  {
            class MyInterfaceImpl implements MyInterface {
                private MyInterfaceImpl(/*... параметры конструктора... */) {
                }
                //…  прочие члены локального внутреннего класса
            }
            return new MyInterfaceImpl (/*…*/);
    }
    else {
            return null;
    }
}
```

Анонимные внутренние классы

- **Анонимные классы встречаются в программах сравнительно часто – когда нужно единожды инстанциировать локальный класс (и этому классу можно не давать имени).**
- *Это не значит, что будет создан только один инстанс этого класса (например, такое инстанциирование локального класса может быть внутри цикла...)*

```
public interface
RandomNumberListener {
    void numberGenerated (int n);
}
```

```
public class OddNumbersLover {
private int myBestNumber = 1;
        private void setBestNumber(int n){
                //…
        }

public RandomNumberListener
getNumberListener ( ) {
        return new RandomNumberListener ( ) {
    Object someMember;
    { /* например, инициализатор, ... */ }

            public void numberGenerated(int n) {
              setBestNumber(n);
            }
          } ;
        }
}
```

- В анонимном классе вместо интерфейса можно использовать базовый класс и его конструктор с параметрами, например:

```
public interface
RandomNumberListener {
    void numberGenerated (int n);
}


public class
RNLImpl implements RandomNumberListener
{
   String preffix;
   public RNLImpl (String pr) {
      preffix = pr;
   }
   public void numberGenerated (int n) {
      System.out.println (preffix + n);
   }
}
```

```
public class OddNumbersLover {
     private int myBestNumber = 1;
           private void setBestNumber( int n){
              //…
           }
     public
     RandomNumberListener getNumberListener( ) {
        return new RNLImpl ( “OGOGO”) {
     public void numberGenerated (int  n) {
        super.numberGenerated (n);
              setBestNumber (n);
           }
        } ;
     }
}
```

- Иметь свой конструктор в анонимном классе нельзя (так как у него нет имени), но можно иметь instance initializer:

```
public class OddNumbersLover {
        private int myBestNumber = 1;
        private void setBestNumber( int n) {
                // …
        }
        public RandomNumberListener getNumberListener( ) {
            return new RNLImpl ( “OGOGO”) {
    private int count;
    {
        count = myBestNumber;
    }
    public void numberGenerated (int  n) {
        super.numberGenerated (n);
                setBestNumber (n);
        count++;
            }
          } ;
        }
}
```

- Анонимные внутренние классы могут либо наследовать класс, либо реализовывать интерфейс (причем - единственный). При этом нет возможности иметь конструкторы, но можно иметь инстанс-инициализатор(ы).
- Локальные и анонимные классы могут использовать локальные переменные / параметры из охватывающего блока / метода только если они не изменяются: если они указаны **final** (до JDK8) или они являются ***effectively final*** (с JDK8 (+) компилятор сам видит их неизменность). Это мы важное требование мы обсудим на семинарах.
- Вложенные в интерфейс классы автоматически являются общедоступными статическими вложенными классами (**static** и **public**):

```
public interface MyInterface {
    class NestedClass  /* это public static class (по определению...)*/ {…}
}
```

Q&A

- **Читать:**
  - **Horstman т.1 (Edition 11), Chapter 6 (Interfaces, Inner Classes):**
    - **п. 6.1 (Interfaces)**
    - **п.6.3 (Inner Classes)**

- Полезные тонкости по всем аспектам Java-программирования – см. в Smart LMS в папке “Useful”:
  - Java. Notes for Professionals