Core Java, осень

Абстракция

Абстракция означает разработку классов исходя из их интерфейсов и функциональности, не принимая во внимание реализацию деталей.

Преимущества

- Применяя абстракцию, мы можем отделить то, что может быть сгруппировано по какому-либо типу.
- Часто изменяемые свойства и методы могут быть сгруппированы в отдельный тип, таким образом основной тип не будет подвергаться изменениям. Это усиливает принцип ООП: *«Код должен быть открытым для Расширения, но закрытым для Изменений»*.
- Абстракция упрощает представление дочерних классов

Абстрактный класс

```
public abstract class Car {

    private String model;
    private String color;
    private int maxSpeed;

    public abstract void gas();

    public abstract void brake();

    // getter(), setter()
}
```

Интерфейс

- Описывает поведение объектов, которые его реализуют
- Не определяет внутреннее состояние
- Определяет двусторонний контракт, выдвигая:
  - требования к реализации
  - требования к пользователям

Требования интерфейса выражаются в:

- Сигнатурах методов (проверяется компилятором)
- Аннотациях (проверяется процессором аннотаций, линтером и тд)
- Документации (проверяется программистом)

Интерфейс

```java
public interface Messenger {

 void sendMessage();

 void getMessage();
}
```

```java
public class Telegram implements Messenger {

 @Override
 public void sendMessage() {

   System.out.println("Отправляем сообщение в Telegram!");
 }

 @Override
 public void getMessage() {
   System.out.println("Читаем сообщение в Telegram!");
 }
}
```

Функциональный интерфейс

Функциональный интерфейс - интерфейс, содержащий один абстрактный метод. Это единственное условие, поэтому static и default методов может быть сколько угодно

```
import java.lang.FunctionalInterface;

@FunctionalInterface
public interface MyInterface{
  // один абстрактный метод
  double getValue();
}
```

`@FunctionalInterface`

Функциональный интерфейс

Аннотация `@FunctionalInterface` нужна для проверки на этапе компиляции, является ли интерфейс функциональным. Если нет, генерируется ошибка.

Функциональный интерфейс

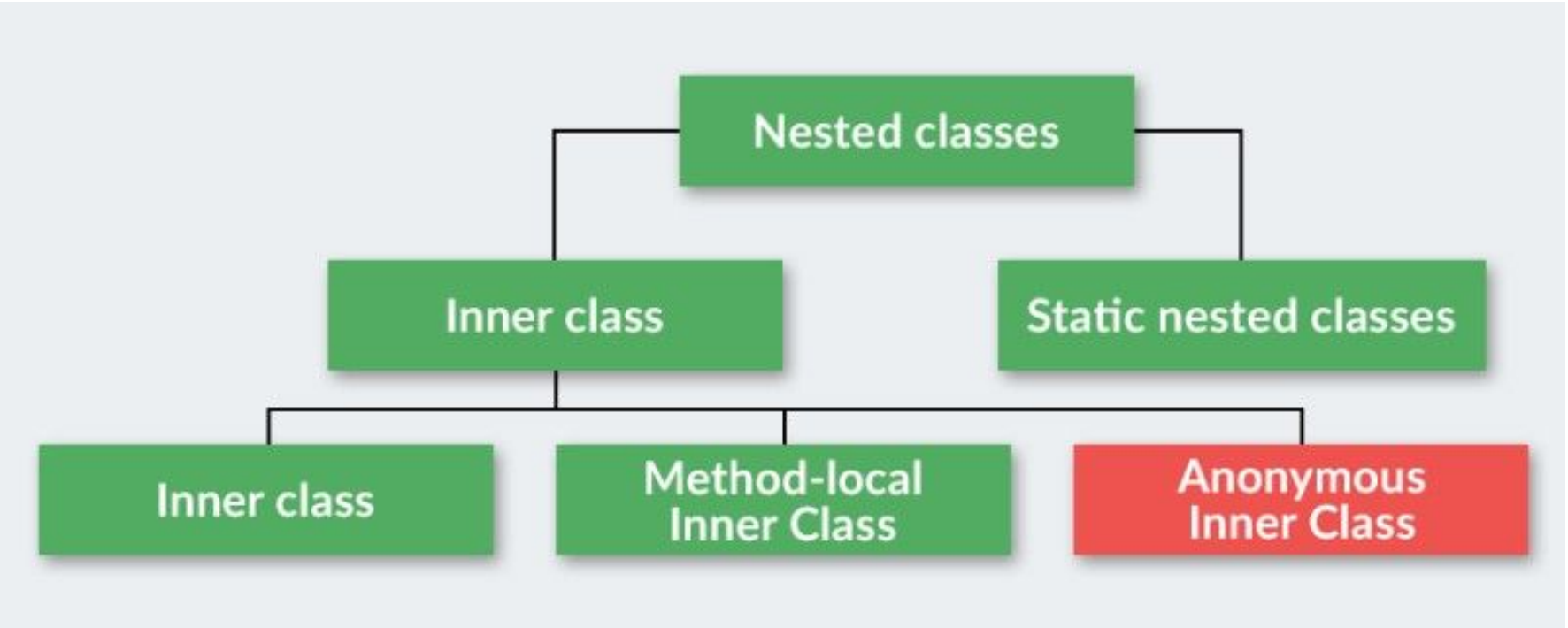
Существует одно исключение касательно того, что функциональный интерфейс должен содержать один абстрактный метод. Исключения составляют методы класса ***Object.***

```
@FunctionalInterface
public interface Generator {

 int next();

 boolean equals(Object o); // тут не будет ошибки
}
```

Анонимный класс



Лямбда-выражения

```
(parameter list) -> lambda body
```

```
public static void main(String[] args) {
 new Thread(() -> System.out.println("Hello")).start();
}
```

Лямбда-выражения

- Если лямбда-функция используется в качестве статического поля, то она имеет доступ ко всем статическим членам этого класса.
- Если лямбда-функция реализована в обычном методе, то она имеет доступ ко всем членам объемлющего класса, если в статическом, то только к статическим.
- Внутри метода лямбда-функция имеет доступ только к ***final*** и ***effective-final*** локальным переменным.

Ссылки на методы

Оператор ::

Ссылка на статический метод	***ContainingClass::staticMethodName***	Все параметры передаются методу. Напр. ***Objects::isNull ~ Objects.isNull(x)***
Ссылка на нестатический метод конкретного объекта	***containingObject::instanceMethodName***	Метод вызывается для заданного объекта Напр. ***System.out::println ~ x -> System.out.println(x)***
Ссылка на нестатический метод любого объекта конкретного типа (класса)	***ContainingType::methodName***	Первый параметр - получатель, остальные - параметры. Напр. ***String::compareToIgnoreCase ~ (x, y) -> x.compareToIgnoreCase(y)***
Ссылка на конструктор	***ClassName::new***	

Ps. Во внутреннем классе можно указать ссылки ***this, super*** на внешний класс следующим образом: ***Объемлющий класс::this::метод***