Принципы S.O.L.I.D.

> *важный совет: запоминайте принципы по буквам, S - Single responsibility, O - open-closed и так далее. Сильно упрощает жизнь.*

`SOLID - это принципы дизайна классов`

1. Single responsibility – Принцип единой ответственности

> Никогда не должно быть больше одной причины изменить класс. <

```
Простой пример - у меня есть класс Moon и класс SpaceShip.
Если у объекта класса Moon будет метод land(), то это будет явное нарушение принципа
единой ответственности, так как этот метод должен быть у экземпляра класса SpaceShip.
У каджого должна быть своя роль и перемешивать эти роли категорически запрещено.
```

2. Open-closed – Принцип открытости/закрытости

> Классы, модули, функции и прочие программные сущности должна быть открыты для расширения, но закрыты для изменения. <

> *Это означает, что должна быть возможность изменять внешнее поведение класса, не внося физические изменения в сам класс.*
> *Следуя этому принципу, классы разрабатываются так, чтобы для подстройки класса к конкретным условиям применения было достаточно расширить его и переопределить некоторые функции.*

```
Яркий пример - поля данных private + сеттеры и геттеры.
```

3. Liskov substitution – Принцип подстановки Барбары Лисков

> Это вариация принципа открытости/закрытости, о котором говорилось ранее. Его можно описать так: объекты в программе можно заменить их наследниками без изменения свойств программы. <

> *Это означает, что подклассы должны переопределять методы базового класса так, чтобы не нарушалась функциональность с точки зрения клиента.*

```
Важный пример: есть базовый класс Laptop, там есть метод setName(), который
устанавлиает поле name (this.name = name)
Есть подкласс класса Laptop - HuaweiMatebook16
и этот подкласс делает @Override сеттера и добавляет строчку this.power = power
Таким образом, если я захочу заменить объект Laptop объектом HuaweiMatebook16, то при
использовании сеттера setName() я случайно подредактирую ещё и поле power. Получается явное нарушение принципа
подстановки Барбары Лисков, так как объектом подкласса нельзя воспользоваться вместо объекта базового класса.
```

4. Interface segregation – Принцип разделённости интерфейсов

> Клиент не должен быть вынужден реализовывать методы, которые он не будут использовать <

```
Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более ме
лкие и специфические, чтобы клиенты мелких интерфейсов знали только о методах, необходимых в работе. В итоге, п
ри изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.
```

> *Кароч, нельзя в одном интерфейсе писать 100500 методов, если они уж совсем по логике блико не связаны.*
> *К примеру, нальзя создавать интерфейс doThings и в нём писать reflect() light() run() sleep() eat() see()*
> *Надо отдельно создать интерфейс для see(), отдельно для eat() и так далее.*

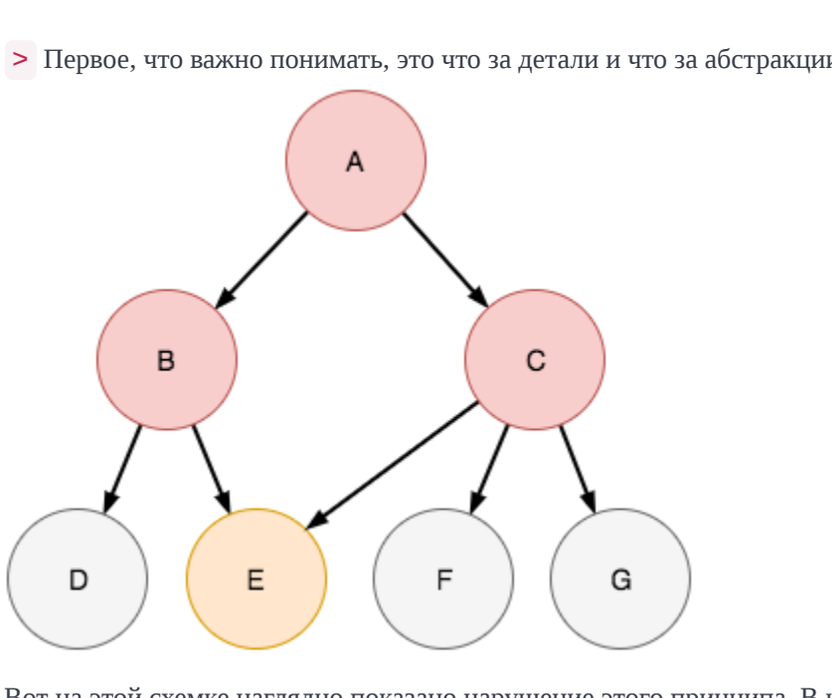
> *Ещё хороший пример с javarush: есть интерфейс generate:*
> *там есть два метода generatePDF() и generatePNG()*
> *а я хочу воспользоваться этим интерфейсом исключительно для создания PDF.*
> *Устроит ли меня такая функциональность? Нет. Я не хочу переопределять два метода.*

5. Dependency inversion – Принцип инверсии зависимостей

> Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. <

```
Программное обеспечение нужно разрабатывать так, чтобы различные модули были автономными и соединялись друг с д
ругом с помощью абстракции.`
```

> Первое, что важно понимать, это что за детали и что за абстракции. Представим такой рисуночек:

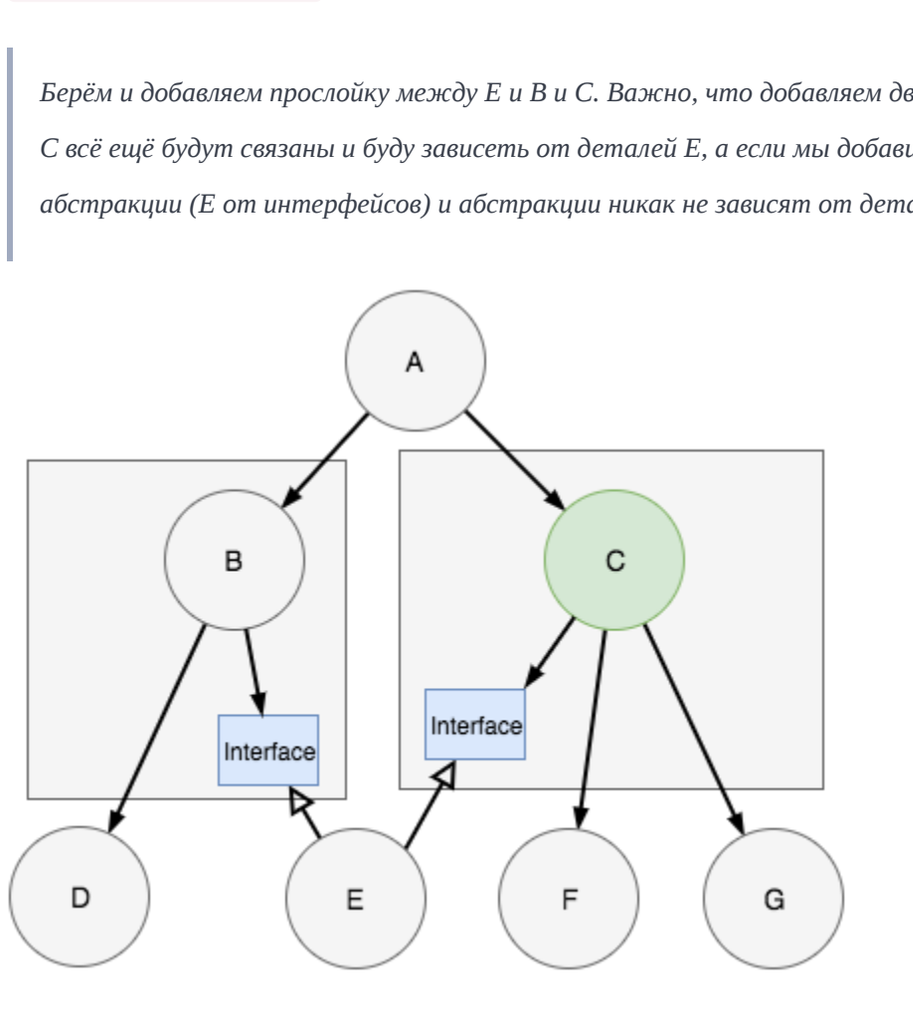


Вот на этой схемке наглядно показано нарушение этого принципа. B и C завязываются на одной какой-то реализации E.

> *Вот представим ситуацию, мы захотели добавить какой-то функционал в реализации E. Поменяли, потом выясняется, что из-за этого что-то сломалось в B и C. И в итоге задача расширения функционала в E превращается в переписывание всей архитектуры.*

`Однако, выход есть`

> *Берём и добавляем прослойку между E и B и C. Важно, что добавляем два интерфейса, а не один, потому что если мы добавим один интерфейс, то B и C всё ещё будут связаны и буду зависеть от деталей E, а если мы добавим два интерфейса - при B и при C, то получается, что детали зависят от абстракции (E от интерфейсов) и абстракции никак не зависят от деталей (B и C никак не зависят от E).*



> Важно понимать, почему в названии принципа есть слово ИНВЕРСИЯ. Потому что мы инвертируем зависимости. Посмотрите на направление стрелочек. Теперь детали зависят от абстракций - E зависит от интерфейсов и вынуждено их реализовывать. А абстракции не от чего не зависят.

> *Ещё пример: есть класс order. Он зависит от двух классов PackageDelivery и PackageRepository.*
> *сейчас класс зависит от других классов. Давайте сделаем, чтобы он зависил от абстракций, как это требуется по принципу инверсии зависимостей.*
> *Создадим два интерфейса OrderDelivery и OrderRepository и засунем в эти интерфейсы уже готовые для этого классы.*
> *и теперь имплементируем эти интерфейсы в классе order. Всё, готово, теперь order зависит от абстракций, а не от классов.*

Класс Object

> *Фактически все классы наследуются от класса Object*
> *Все остальные классы, даже те, которые мы добавляем в свой проект,*
> *являются неявно производными от класса Object.*
> *Поэтому все типы и классы могут реализовать те методы, которые определены в классе Object*
> *Основные - toString() hashCode() equals(), но ещё есть getClass()*
> *Подробнее про это методы можно почитать m---------y---------m*

toString()

> *возвращает строковое представление объекта. Read here*

hashCode() & equals()

> *for an object returns an integer value, generated by a hashing algorithm. This can be used to identify whether two objects have similar Hash values which eventually help identifying whether two variables are pointing to the same instance of the Object. If two objects are equal from the .equals() method, they must share the same HashCode*
> *equals() по дефолту сравнивает ссылки.*

Ещё раз, для чего нужны эти методы

> *quals() and hashCode() are majorly used to identify whether two objects are the same/equal. Whereas, toString() is used to Serialise the object to make it more readable in logs.*

Контракты equals() и hashcode()

> *это правила переопределения этих медов, они не сложные, но их надо знать и понимать.*
> *почитайте m-y-m, быстрее всё поймёте.*

`Note: !You must override hashCode() in every class that overrides equals()!`

Принципы STUPID

> *достаточно простые принципы, спрашивают не так подробно, как SOLID. Если поняли SOLID, то поняли STUPID.*

1. Singletone: синглтон - повторение одного и того же шаблона
2. Tight coupling: нильная связанность - если изменение одного модуля требует изменения другого, то это плохо.
3. Untestability: невозможность тестирования - следствие сильной связанности и говнокода
4. Premature optimisation: преждевременная оптимизация - система становится сложнее, код - нечитабельным.
5. Indescriptive naming: не дескриптовое присвоение имени - называй классы и всё остальное нормально, а не как конч какой-то.
6. Doubling: дублирование кода - Любой код должен избегать повторений, потому что дублированный код неэффективен.

Почитать подробнее можно тут

Как и где хранятся строки в java и что такое String Pool

> *Строки хранятся в куче. Задать можно двумя способами: интернированием и конструктором.*

```
Чтобы задать строку интернированием, надо записать подряд символы в двойных кавычках: "ThisIsString".
Эта строчка будет храниться в специальном месте кучи, которое носит название *String Pool*.
Если задать строку через конструктор, то она будет просто храниться в куче.
Зачем это? Это способ оптимизации памяти, через конструктор можно создать кучу экземпляров одной и той же строк
и и все объекты этой строки будут ссылаться на разные места в куче.
Если интернированием создавать одну и ту же строку, то все эти объекты будут ссылаться на одно и то же место в
String Pool.
```

! хорошая, подробная и понятная статья !

Особенности наследования и проблема ромба

> *Возьмём 4 класса: супер-класс, класс а, класс б и класс в, который наследует а и б доновременно.*
> *в классе ф есть метод doSomething() и в классе б есть метод doSomething().*
> *какой из них выбрать?)*
> *Решение этой проблемы: композиция. Создать в классе C объекты A и B и переписать для них методы*

```
public class ClassC{

    ClassA objA = new ClassA();
    ClassB objB = new ClassB();

    public void test(){
        objA.doSomething();
    }

    public void methodA(){
        objA.methodA();
    }

    public void methodB(){
        objB.methodB();
    }
}
```

`но множественное наследование работает с интерфейсами`

> *Здесь все идеально, поскольку интерфейсы – это только резервирование/описание метода, а реализация самого метода будет в конкретном классе, реализующем эти интерфейсы, таким образом нет никакой возможности столкнуться с неопределенностью при множественном наследовании интерфейсов.*

Интерфейс

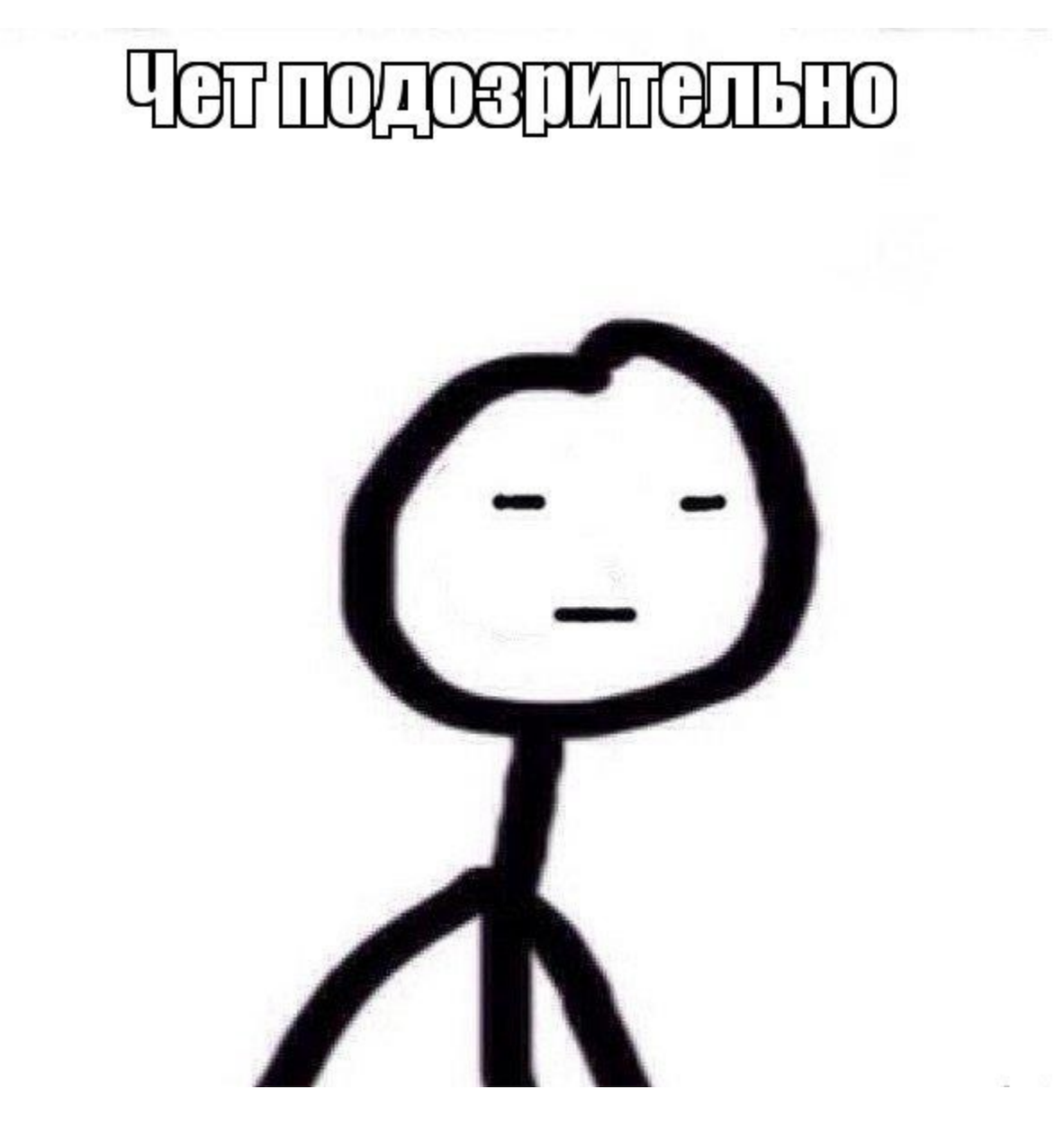
> это абстрактный тип, используемый для описания поведения, которое должны реализовать классы <
> интерфейс описывает поведение, которым должны обладать классы, реализующие этот интерфейс. <
внутри него ни геттеров, ни сеттеров. Можно создавать статические поля и делать статические методы, но переопределять их, очевидно, нельзя.
> Итак, методы могут быть (Java 8): default, static; (Java9): private, private static
> Переменные могут быть только константы public static final (модицикаторы по умолчанию)
`! у интерфейса нет состояния, а у абстрактного класса есть и поведение и состояние`

Абстрактный класс

```
> Использует 4 концепцию ООП - абстракцию. Нужен для того, чтобы объединять свойства и поведение объектов.
> является основой для всех дочерних классов
> может иметь как абстрактные, так и обычные методы
> нельзя создавать объекты; а у  дочерних неабстрактных, очевидно, - можно
```

обманщик

> *Погодите-ка, но что абстрактный класс, что интерфейс - это абстракции. Что там что там есть абстрактные методы, их надо реализовывать при наследовании, оба описывают поведение, да и практик сидит улыбается, ты чё наврал мне?*



Отличие абстрактного класса от интерфейса

> Отличие от интерфейса в том, что интерфейс нужен для описания какого-то универсального поведения. К примеру, интерфейс reflectLight() служит для того, чтобы описывать отражение света. Свет может отражать зеркало, стеклянное здание, поверхность какого-то материала, крыша, озеро, телефон, да всё, что угодно. А абстрактный класс нужен чтобы выделить какие-то схожие свойства у объектов, к примеру, абстрактный класс Human() может выделить общие свойства у всех людей, которые в нашей программе свойственны ТОЛЬКО людям - read(), prepareHomework(), study() и так далее. А также абстрактный класс может выносить getName(), bloodType() и прочие схожие характеристики подклассов класса Human(). Ну и также не стоит забывать, что у интерфейсов разрешено множественное наследование, чего нельзя сказать об абстрактных классах <

Вот тут подробно написано различие вместе с модификаторами доступа и всеми приколами

Enum

это “перечисление”
> просто класс, все возможности обычного класса у него есть. Состоит из набора именованных констант <
> Как это работает. У енама, как и у любого другого класса, есть конструктор. Если мы его не задаём в явном виде, то он создаётся пустой по-дефолту. Когда мы пишем константу, то эта константа - это экземпляр данного класса. То есть, мы можем написать константу и в скобках передать какие-нибудь параметры, затем написать конструктор, который будет принимать эти параметры и ровно так же, как и без скобок - вызовется конструктор, только уже в этот раз другой, который мы только что написали, и создастся объект с названием этой константы. <

В енаме ниже я “перечислил” нужные мне константы. Если я захочу получить объект звезды, я напишу CosmicObjectType.STAR.

```
public enum CosmicObjectType{
    STAR,
    ASTEROID,
    PLANET,
    GALAXY
}
```

В следующем енаме я уже хочу создавать объект, у которого будет какая-то форма. Следовательно, я создаю конструктор для констант и передаю туда какую-то форму объектов.

```
public enum CosmicObjectType{
    String shape;
    public CosmicObjectType(string shape) {
        this.shape = shape
    }
    STAR("star"),
    ASTEROID("dodecahedron"),
    PLANET("sphere"),
    GALAXY("abstractShape")
}
```

Функциональное программирование

— это программирование, в котором функции являются объектами, и их можно присваивать переменным, передавать в качестве аргументов другим функциям, возвращать в качестве результата от функций и т. п

Лямбда-выражения

> *Лямбда-выражение в java — такое синтаксическое выражение, которое состоит из лямбда-оператора посерёдке (стрелочка ->), передаваемых аргументов слева стрелочки и возвращаемых значений(реализации метода) справа. Теперь обо всё попорядочку.*
> *Как это, значит, работает. Чтобы лямбда-выражения работали, им нужен функциональный интерфейс. Определение дам чуть позже, а пока к сути. Функциональный интерфейс содержит в себе какой-то объявленный, но нереализованный метод. Когда мы пишем лямбда-выражение, аргументы, которые мы пишем слева стрелочки - это аргументы того самого метода, который java будет искать в функциональном интерфейсе, а возвращаемые значения, которые мы будем писать справа от стрелочки - это реализация этого метода.*

```
По сути, лямбда выражение реализовывает метод в функциональном интерфейсе. Ничего не напоминает? Анонимный клас
с. Но об этом  вы узнаете в 4 лабе, однако я рекомендую ознакомиться с этим понятием хотя бы поверхностно для б
олее глубокого понимания.
```

Функциональный интерфейс

Это такой интерфейс, в котором должен быть ТОЛЬКО ОДИН нереализованный, но объявленный метод, помимо этого не накладывается ограничения на количество дефолтных и статических методов. Их можно писать сколько угодно.

> Функциональных интерфейсов по умолчанию в джаве много. А именно, 9.
Вот они: Supplier, Consumer, Predicate, BiPredicate, Function, BiFunction, UnaryOperator, BinaryOperator.
Из того, что надо знать: что принимает и что возвращает. Параметры и тип параметров. <

Функциональный интерфейс	Параметры	Возвращаемый тип	Абстрактный метод	Описание
Supplier<T>	0	T	get	Возвращает объект типа T. Содержит метод get()
Consumer<T>	T	void	accept	Выполняет операцию над объектом типа T. Содержит метод accept()
BiConsumer<T, U>	T, U	void	accept	Выполняет операцию заданную в методе accept() над объектами T и U
Predicate<T>	T	boolean	test	Определяет, удовлетворяет ли объект типа T некоторому условию. Возвращает логическое значение, обозначающие результат. Содержит метод test()
BiPredicate<T, U>	T, U	boolean	test	Возвращает логическое значение true, если оба аргумента удовлетворяют условию, заданному методом test(), иначе false
Function<T, R>	T	R	apply	Выполняет операцию над объектом типа T и возвращает в результате объект типа R. Содержит метод apply()
BiFunction<T, U, R>	T, U	R	apply	Выполняет операцию над объектами T и U и возвращает результат R. Содержит метод apply
UnaryOperator<T>	T	T	apply	Выполняет унарную операцию над объектом типа T и возвращает результат того же типа. Содержит метод apply()
BinaryOperator<T, T>	T, T	T	apply	Выполняет логическую операцию над двумя объектами типа T и возвращает результат того же типа. Содержит метод apply()

`Желательно понять что такое дженерики`, потому что тут они используются (вот эти штуки внутри треугольных скобок это <дженерики>)

Пример использования лямбда-выражения

Обход элементов масива в цикле:

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
integers.forEach(item -> System.out.println(item));
```

> *возвращается a+b*

```
(a, b) -> a + b
```

Ссылка на методы.

> *Итак, что же за ссылка на методы такая. По сути, это те же лямбда-выражения, но с ещё более упрощённым синтаксисом.*
> *> Ссылка на метод - это синтаксическая конструкция, которая состоит из оператора двойного двоеточия [::], суть этого оператора в том, чтобы обратиться к методу какого-то класса.*
> *Ссылки бывают четырёх видов:*

```
ContainingClass::staticMethodName # ссылка на статический метод
containingObject::instanceMethodName # ссылка на нестатический метод какого-то конкретного объекта
ContainingType::methodName # ссылка на нестатический метод любого объекта конкретного типа
ClassName::new # ссылка на конструктор
```

Для более глубокого понимания советую прочитать эту статью

Что такое маркерный интерфейс и зачем он нужен

! маркерный интерфейс совершенно пустой, там ничего нет, только название !

> *Это такая тема, которая используется для того, чтобы указывать, что экземпляры какого-то класса МОГУТ что-то ДЕЛАТЬ.*
> *К примеру, интерфейс Serializable имплементируется в какой-то класс, чтобы указать, что экземпляры этого класса могут сериализоваться.*
> *Когда мы передаём эти объекты какому-то другому классу, там есть проверка на наличие этого интерфейса и если он не будет имплементирован, то сериализация не произойдёт.*

Напоминаю, сериализация - это конвертирования объектов в битовую последовательность для последующей передачи по сети и дальнейшщей десериализации, то есть восстановления исходного вида объекта из последовательности битиков.

Литература

> Серия отличных и коротких видосов точечно проходящихся по этим темам: (настоятельно рекомендую к просмотру) плейлист <
> Хорошая серия про лямбды и функциональщину: ссылка <
> Лекции Письмака А.Е.: ссылка <
> Ковarианты возвращаемых типов: ссылка <
> S.O.L.I.D: ссылка <
> S.T.U.P.I.D: ссылка <