Оглавление

[SOLID 1](#_Toc154186345)

[STUPID 2](#_Toc154186346)

[Object 2](#_Toc154186347)

[Методы: 2](#_Toc154186348)

[Поля: 5](#_Toc154186349)

[Особенности реализации наследования в Java. Простое и множественное наследование. 6](#_Toc154186350)

[Понятие абстрактного класса. Модификатор abstract. 8](#_Toc154186351)

[Понятие интерфейса. Реализация интерфейсов в Java, методы по умолчанию. Отличия от абстрактных классов. 9](#_Toc154186352)

[Перечисляемый тип данных (enum) в Java. Особенности реализации и использования. 10](#_Toc154186353)

[Методы, поля и классы с модификаторами static и final. 11](#_Toc154186354)

[Static 11](#_Toc154186355)

[Final 12](#_Toc154186356)

[Перегрузка и переопределение методов. Коварианты возвращаемых типов данных 13](#_Toc154186357)

[Элементы функционального программирования в синтаксисе Java. Функциональные интерфейсы, лямбда-выражения. Ссылки на методы. 13](#_Toc154186358)

# SOLID

**1 Единственная обязанность (Single Responsibility):**

У класса должна быть только та обязанность которая свойственна объектам типа этого класса, например у класса человек, наряду с методом eat() не должно быть метода createOcean()

**2 Открытость-закрытость (Open-closed)**

Классы должны быть закрыты для изменений и открыты для расширения.  
То есть новая функциональность вводится путём добавления нового кода, а не путём изменения уже имеющегося, через наследования, чтобы наследник менял метод родителя.   
Хорошим «тоном» также является для полей создавать геттеры и сеттеры

**3 Принцип постановки Барбары Лисков (Liskov Substitution Principle)**

В любой точке подстановки наследника класса, его исполнение должно происходить согласно ожиданиям, соответствовать поведению родителя, например, если есть наследник ToyDuck от класса Duck, который имеет заряд и метод swim() который будет работать только если есть заряд, то при вызове массива из игрушечных уток и обычных уток может произойти ошибка если у игрушечной не будет заряда, также обычно слово extends в наследовании, часто заменяется такой формулировкой: ToyDuck is a Duck, поэтому мы не можем подставить понятия только схожие по виду, ведь игрушечная утка это всё же не утка. Если всё же такое наследование нужно, то можно в обход соблюсти этот принцип, например сделав другой метод который проверяет может ли утка плавать.

**4 Разделение интерфейсов (Interface Segregation Principle)**

Нужно разделять интерфейсы по смыслу, то же самое, что первый принцип с классами, но с интерфейсами. Если интерфейс относится и к одной и к другой сущности, то его нужно разделить на два.

**5 Инверсия зависимостей (Dependency Inversion Principle)**

Принцип DI гласит, что высокоуровневые модули не должны зависеть от низкоуровневых модулей, оба типа модулей должны зависеть от абстракций. Кроме того, абстракции не должны зависеть от деталей, детали должны зависеть от абстракций.

# STUPID

**Singleton (Одиночка):** Избегайте чрезмерного использования шаблона проектирования "Одиночка" (Singleton), так как он может создавать проблемы с тестированием и поддержкой.

**Tight Coupling (Тесная связь):** Избегайте тесной связи между компонентами программы. Чем менее зависимыми являются компоненты друг от друга, тем легче изменять и тестировать код.

**Unnecessary Complexity (Ненужная сложность):** Избегайте добавления избыточной сложности в код, которая не приносит явных преимуществ. Простота обычно предпочтительнее.

**Premature Optimization (Преждевременная оптимизация):** Не оптимизируйте код до тех пор, пока не станет ясно, что это действительно необходимо. Преждевременная оптимизация может привести к усложнению кода без реальной выгоды.

**Indescriptive Naming (Неописательные имена):** Избегайте использования неясных, неописательных имен переменных, классов и методов. Читаемый код облегчает понимание и поддержку программы.

**Duplication (Дублирование):** Избегайте дублирования кода. Дублирование усложняет поддержку и изменение программы, и может привести к ошибкам из-за несогласованности.

# Object

Класс Object является базовым строительным блоком для всех объектов в Java, и его методы предоставляют базовую функциональность для работы с объектами, такую как сравнение, хеширование и управление потоками.

## Методы:

**equals(Object obj):**

У класса Object есть своя реализация метода equals, которая просто сравнивает ссылки:  
Возвращает true, если текущий объект равен объекту obj.  
По умолчанию реализация в классе Object сравнивает ссылки на объекты.

**hashCode():**

hashCode нужен для быстрого сравнения.  
hashCode() для каждого объекта возвращает определенное число. Какое именно – это тоже решает разработчик класса, как и в случае с методом equals. Этот метод используется, когда объекты добавляются в хеш-таблицы, такие как HashMap.  
Если переопределен метод equals(), рекомендуется также переопределить hashCode().

у двух разных объектов может быть одинаковый hashCode (разные люди могут жить в одной общаге)

у одинаковых объектов (с точки зрения equals) должен быть одинаковый hashCode.

хеш-коды должны быть выбраны таким образом, чтобы не было большого количества различных объектов с одинаковыми hashCode.

HashMap — структура данных, одна из коллекций языка Java. Представляет собой хэш-таблицу. Так называется набор из пар «ключ-значение», где у ключей есть хэши, то есть числовые уникальные идентификаторы. Они высчитываются для каждого ключа.

**toString():**

Возвращает строковое представление объекта. Обычно используется для вывода на консоль или в логи.  
По умолчанию возвращает строку, содержащую имя класса и хеш-код объекта в формате "имя\_класса@хэш\_код"

**getClass():**

Возвращает объект класса Class, представляющий текущий объект.

Класс Class в языке программирования Java является частью пакета java.lang и представляет собой объект, который содержит метаинформацию о классе. Каждый объект в Java имеет свой собственный Class объект, который можно использовать для получения информации о типе объекта во время выполнения программы. Class тоже наследуется от Object. Однако, не все аспекты класса могут быть доступны или могут быть доступны с ограничениями:

Приватные члены:

Методы getDeclaredMethods(), getDeclaredFields(), getDeclaredConstructors() предоставляют доступ к всем членам класса, включая приватные. Однако, для получения доступа к приватным членам, вам может потребоваться использовать методы setAccessible(true) из пакета java.lang.reflect, что может вызвать проблемы с безопасностью.

Обобщенные типы:

Информация об обобщенных типах (generic types) в Java утрачивается во время   
компиляции из-за стирания типов. Поэтому при использовании рефлексии нельзя полностью восстановить обобщенные типы.

Аннотации:

Хотя можно получить доступ к аннотациям через методы getAnnotations(), getDeclaredAnnotations(), getAnnotation(Class annotationClass), нельзя получить информацию о том, как аннотации были заданы в исходном коде (например, значения аннотаций, заданные в круглых скобках).

Локальные переменные:

Рефлексия не предоставляет прямого доступа к локальным переменным метода, так как они удаляются во время компиляции.

Структуры управления доступом и полиморфизм:

Рефлексия может проигнорировать модификаторы доступа и полиморфизм. Например, можно получить доступ к приватным членам, а также вызвать методы, даже если они переопределены в подклассах.

**clone():**

Класс Object в Java содержит метод clone(), который создает и возвращает поверхностную копию объекта.   
Однако, по умолчанию этот метод реализован так, что он выполняет клонирование только видимых примитивных полей объекта, и клонирует ссылки на другие объекты не приметивных типов.  
Клонирование объекта в классе Object реализовано очень примитивно – при клонировании создается всего один новый объект: просто создается еще один объект и его полям присваиваются значения полей объекта-образца.

Если копируемый объект содержит ссылки на другие объекты, то ссылки будут скопированы, дубликаты тех объектов не создаются.

Интерфейс Cloneable – это так называемый интерфейс-маркер, который не содержит никаких методов. Он используется, чтобы маркировать (помечать) некоторые классы.  
При вызове метода clone(), Java проверяет, был ли у объекта интерфейс Cloneable. Если да — клонирует объект методом clone(), если нет — выкидывает исключение CloneNotSupportedException.

**notify(), notifyAll(), wait():**

Используются для управления многопоточностью. notify() и notifyAll() используются для синхронизации потоков, а wait() заставляет поток ожидать до тех пор, может использоваться только пока объект занят, то есть находится внутри блока synchronized пока другой поток не вызовет notify() – снимет случайный поток с ожидания, или notifyAll() – снимет все потоки с ожидания. Если важно разблокировать определенный поток, то рекомендуется использовать notifyAll(), чтобы разбудить все потоки, ожидающие на объекте. Затем каждый поток должен проверить условие, по которому он ожидал, и решить, должен ли он продолжить выполнение или вернуться в ожидание.

Есть еще модификации метода wait():

|  |  |
| --- | --- |
| **void** wait(**long** timeout) | Нить «замерзает», но через переданное количество  миллисекунд автоматически «размораживается». |
| **void** wait(**long** timeout, **int** nanos) | Нить «замерзает», но через переданное количество  миллисекунд и наносекунд автоматически  «размораживается». |

**finalize():**

Этот метод вызывается перед тем, как объект будет удален сборщиком мусора. Он может быть переопределен для выполнения необходимых действий перед уничтожением объекта  
Предназначен для автоматического освобождения системных ресурсов, занимаемых объектом, на котором будет данный метод вызван. Это кажется удобным, чтобы не помнить постоянно, например, что мы должны закрыть соединение с каким-то ресурсом, когда оно больше не требуется.

Стоит отметить, что в очередь на финализацию попадают только те объекты, в которых переопределен метод finalize.

finalize() можно использовать только в двух случаях:

1. Проверка/подчистка ресурсов с логированием.

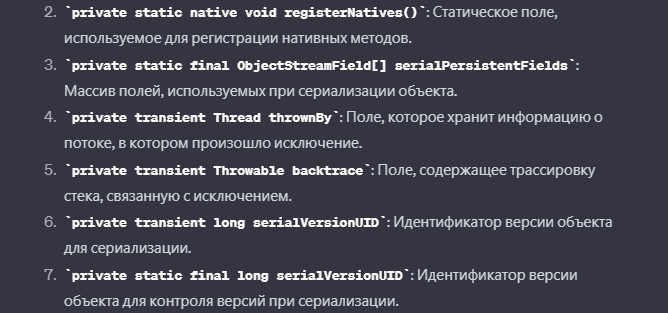
2. При работе с нативным кодом, который не критичен к утечке ресурсов.

finalize() замедляет работу GC по очистке объекта в 430 раз

## Поля:

**public final Class<?> getClass():**

Поле class предоставляет объект класса Class, представляющий тип объекта.



Native методы в Java представляют собой методы, написанные на других языках программирования

Сериализация - это процесс преобразования объекта в последовательность байтов, которую затем можно сохранить в файле или передать по сети, и в дальнейшем восстановить обратно в объект

Трассировка стека (stack trace) представляет собой список вызовов методов (фреймов стека), который выводится в момент возникновения ошибки или исключения в программе. Каждый элемент трассировки стека указывает на метод в стеке вызовов, начиная с места, где произошла ошибка, и вплоть до точки, где программа была запущена.

# Особенности реализации наследования в Java. Простое и множественное наследование.

При создании объекта класса-наследника в Java происходит несколько шагов:

**Выделение памяти:** Создается блок памяти для хранения данных нового объекта. Этот блок включает в себя место для всех полей объекта, включая унаследованные от родительского класса.

**Инициализация полей:** Выполняется инициализация полей нового объекта. Это включает в себя установку значений для всех полей, как определено в классе-наследнике и его родительском классе.

**Вызов конструктора:** Вызывается конструктор класса-наследника. Если в классе-наследнике отсутствует явно определенный конструктор, будет вызван конструктор по умолчанию (без параметров). Если в классе-наследнике определен конструктор, то он может вызвать конструктор родительского класса с использованием ключевого слова super и передать необходимые параметры.

При вызове конструктора наследника в Java по умолчанию вызывается конструктор родительского класса. Это происходит автоматически, и если в классе-наследнике не указан явно вызов конструктора родительского класса с использованием ключевого слова super, то будет вызван конструктор по умолчанию (без параметров) родительского класса.

Ключевое слово super используется для обращения к членам (полям и методам) родительского класса из дочернего класса. Использование super может происходить через точечную нотацию (super.член) и в скобках при вызове конструктора родительского класса (super(параметры)).

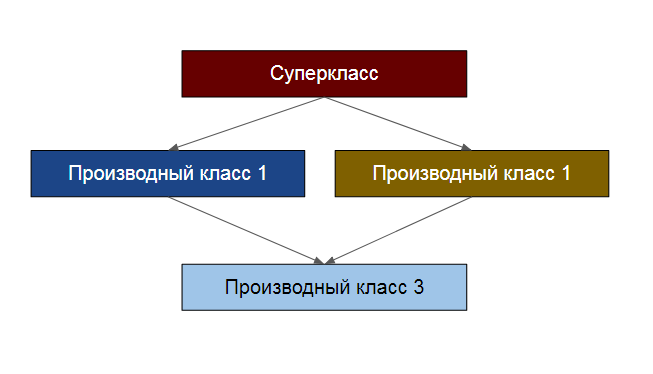
В Java поддерживается только одиночное наследование классов, что означает, что каждый класс может иметь только один прямой родительский класс. Однако, множественное наследование реализуется через интерфейсы.

**Множественное наследование через интерфейсы в Java:**

Множественное наследование может быть реализовано с использованием интерфейсов. Интерфейс в Java представляет собой абстракцию, предоставляющую набор методов, которые подклассы должны реализовать.

**«Проблема Алмаза»**

Для более простого понимания проблемы алмаза допустим, что множественное наследование поддерживается в Java. В этом случае, мы можем получить классы с иерархией показанной на рисунке ниже.



Предположим ,что SuperClass — это абстрактный класс, описывающий некоторый метод, а классы ClassA и ClassB — реальные классы.   
"реальные классы" относиться к классам, объекты которых могут быть созданы напрямую, и они представляют конкретные сущности или объекты в программе

В ООП сущностью может считаться объект, который представляет экземпляр класса.  
Общепринято считается, что объект - это конкретный экземпляр класса, в то время как термин "сущность" может подчеркивать более абстрактное представление объекта, часто связанное с объектно-ориентированным проектированием или сущностями данных.

Возвращаясь к проблеме Алмаза   
Допустим Суперкласс – абстрактный, и он реализует метод dosomething()

Производные классы 1 и 2 переопределяют его.

Допустим в производном классе 3 есть метод test(), в теле которого вызывается метод dosomething() родительского класса, что и приведет к неопределенности, так как компилятор не знает о том, метод какого именно суперкласса должен быть вызван. Благодаря очертаниям диаграммы наследования классов в этой ситуации, напоминающим очертания граненого алмаза проблема получила название «проблема Алмаза». Это и есть основная причина, почему в Java нет поддержки множественного наследования классов.

# Понятие абстрактного класса. Модификатор abstract.

**Абстрактный класс (Abstract Class):**

Абстрактный класс объявляется с использованием ключевого слова abstract.

Он может содержать как обычные (некоторые могут быть реализованы в самом абстрактном классе), так и абстрактные методы (методы, которые объявлены, но не реализованы).

Абстрактный класс может иметь конструктор, и он может быть расширен классами наследниками.

**Абстрактный метод (Abstract Method):**

Абстрактный метод объявляется внутри абстрактного класса без фактической реализации.

Класс, наследующий абстрактный класс, должен предоставить реализацию для всех абстрактных методов или сам быть объявлен как абстрактный.

**Модификатор abstract:**

Когда класс содержит хотя бы один абстрактный метод, сам класс должен быть объявлен как абстрактный с использованием модификатора abstract.

Абстрактные методы также могут содержать модификаторы доступа (public, private, protected), но они не могут быть объявлены как static или final.

Почему так?

**Static (статические):** Методы, объявленные как статические, принадлежат классу, а не объекту. Они могут быть вызваны без создания экземпляра класса. Абстрактные методы связаны с экземплярами классов и их подклассов, их реализация зависит от конкретного экземпляра класса. Статические методы не имеют доступа к состоянию экземпляра, поэтому объявление абстрактного метода как статического не имело бы смысла.

**Final (финальные):** Методы, объявленные как финальные, не могут быть переопределены в подклассах. Абстрактные методы создаются с целью предоставить общий интерфейс для всех подклассов и оставить реализацию для них. Если абстрактные методы могли бы быть объявлены как финальные, это ограничило бы возможность их реализации в подклассах, нарушая цель абстракции и наследования.

**Interface по умолчанию подразумевает абстракцию**

**Наследование от абстрактного класса:**

Класс, наследующий абстрактный класс, должен предоставить реализацию для всех абстрактных методов.

Если он этого не делает, то сам должен быть объявлен как абстрактный.

# Понятие интерфейса. Реализация интерфейсов в Java, методы по умолчанию. Отличия от абстрактных классов.

**Понятие интерфейса:**

Интерфейс в программировании — это абстрактный тип данных, предоставляющий соглашение о том, какие методы должны быть реализованы классами, которые поддерживают данный интерфейс. Интерфейсы обеспечивают абстракцию и стандартизацию взаимодействия между классами. В Java интерфейс объявляет сигнатуры методов, но не содержит их реализации.

**Реализация интерфейсов в Java:**

Чтобы класс реализовал интерфейс в Java, он должен использовать ключевое слово implements. И предоставить свою реализацию всех абстрактных методов.

Использование аннотации @Override не обязательно, но рекомендуется. Эта аннотация позволяет компилятору проверить, действительно ли метод переопределяет метод из суперкласса или интерфейса. Если вы помечаете метод как @Override, и он не переопределяет ни один метод из суперкласса или интерфейса, компилятор выдаст ошибку.

Методы по умолчанию (Default Methods):

Они позволяют добавлять новые методы в интерфейсы без необходимости изменения кода всех классов, реализующих этот интерфейс.

Пример:

public interface MyInterface {  
 void myMethod();  
  
 default void defaultMethod() {  
 // реализация метода по умолчанию  
 }  
}

Класс, который реализует этот интерфейс может выбрать, переопределить метод по умолчанию или использовать его как есть.

**Отличия от абстрактных классов:**

**Множественное наследование интерфейсов:** Класс может реализовывать несколько интерфейсов, что не является возможным для абстрактных классов.

**Поля в интерфейсах:** Интерфейсы могут содержать только статические константы (public static final), тогда как абстрактные классы могут иметь поля с состоянием.

**Конструкторы:** Интерфейсы не могут иметь конструкторов, в то время как абстрактные классы могут.

В целом, интерфейсы предоставляют более гибкий механизм для реализации множественного наследования и определения стандартных контрактов, в то время как абстрактные классы обеспечивают более сильную связь с конкретными реализациями и могут содержать состояние объекта.

# Перечисляемый тип данных (enum) в Java. Особенности реализации и использования.

Еnum (перечисляемый тип данных) в Java представляет собой специальный тип, который представляет собой набор констант. Он предоставляет удобный способ определения и использования фиксированного набора значений в коде.

Обращение к данным происходит через точечную нотацию формата НазваниеEnum.НазваниеКонстанты

Перечисления в Java могут содержать конструкторы, методы и даже реализовывать интерфейсы

В перечислении (enum) в Java, каждая константа представляет собой экземпляр перечисления и, как правило, является public static final

**Основные особенности enum в Java:**

**Ограничение на конструкторы:** Конструктор в перечислении должен быть private. Это ограничение является частью стандарта Java для перечислений и гарантирует, что экземпляры перечисления создаются только один раз.

**Ограничение на сравнение:** Перечисления по умолчанию сравниваются по их позиции в списке. Для сравнения по значениям или другим критериям потребуется переопределение метода compareTo().

**Ограниченный и фиксированный набор значений:** Все константы enum известны на этапе компиляции, и их количество фиксировано.

**Определение методов и полей:** enum может содержать методы, конструкторы и поля, что позволяет добавлять к каждой константе дополнительные свойства.

**Использование в switch-выражениях:** enum может использоваться в конструкциях switch, что делает код более читаемым и устойчивым к ошибкам.

**Безопасность типов:** Переменные, объявленные с использованием enum, могут принимать только значения, определенные в самом enum, что повышает безопасность типов. Переменные, объявленные с использованием перечисления, могут принимать только определенные значения, которые являются константами этого перечисления.

**Статический метод values():** В enum автоматически генерируется статический метод values(), который возвращает массив всех констант в том порядке, в котором они были объявлены.

Обеспечивает удобный способ работы с ограниченным набором констант.

**Методы enum**

**name():** Возвращает имя перечисления в виде строки. Этот метод является final и не может быть переопределен.

**ordinal():** Возвращает порядковый номер константы в перечислении, начиная с 0. Также является final

**values():** Статический метод, возвращающий массив всех значений перечисления в том порядке, в котором они объявлены.

**valueOf(String name):** Статический метод, возвращающий константу перечисления по её имени.

**compareTo(Enum<E> o):** Метод сравнения для упорядочивания констант в перечислении. Этот метод также является final.

**getDeclaringClass():** Возвращает класс, в котором была объявлена константа.

В Java символы <> также используются для обозначения параметров типа в контексте использования дженериков в методах. Если внутри этих символов стоит вопрос, это может означать использование wildcard (знак вопроса с пробелом: ?) или ограниченного типа.

**Wildcard (знак вопроса с пробелом - ?):**

<?> обозначает wildcard, что позволяет использовать любой тип.

**Ограниченный тип (bounded type):**

Если вопрос используется с ключевыми словами extends или super, это означает ограничение сверху или снизу.

# Методы, поля и классы с модификаторами static и final.

## Static

**Статические переменные (static fields):**

Переменные, объявленные с ключевым словом static, относятся к классу, а не к экземпляру объекта.

Они создаются только один раз при загрузке класса в память и используются всеми экземплярами этого класса.

Обращение только через имя класса, а не через экземпляр объекта.

Поля и методы, объявленные с модификатором static, видны везде, где доступен класс или интерфейс, которому они принадлежат. Такие элементы принадлежат не конкретному экземпляру класса, а самому классу, и поэтому они доступны на уровне класса вне зависимости от создания экземпляров этого класса.

**Статические методы (static methods):**

Методы, объявленные с ключевым словом static, принадлежат классу, а не конкретному объекту.

Можно вызывать без создания экземпляра класса.

**Статический блок кода (static block):**

Блок кода, выполняемый при загрузке класса в память.

Выполняется только один раз, независимо от количества созданных экземпляров класса.

**Статичные классы:**

Класс, объявленный с ключевым словом static, считается статичным.

Статичные классы могут содержать только статичные методы и переменные.

Экземпляр статичного класса создавать нельзя.

**Ограничения static**

**Статические переменные (static fields):**

Одно значение для всех экземпляров класса. Если изменить значение в одном месте, изменения отразятся во всех экземплярах.

**Статические методы (static methods):**

Нельзя использовать ключевое слово this в статическом методе, потому что он не привязан к конкретному экземпляру объекта.

**Общее**

Статические элементы могут быть сложны для тестирования, так как они имеют глобальное состояние.

Использование статических элементов может привести к проблемам с многопоточностью, так как они являются общими для всех потоков.

## Final

**Финальные переменные (final variables):**

Внутри анонимных классов (или лямбда-выражений), которые могут использовать локальные переменные, можно использовать только финальные переменные или те, которые эффективно финальны (то есть их значение не меняется после первоначальной инициализации).

Должны быть проинициализированы перед тем, как будет завершено создание объекта (в конструкторе или при объявлении).

Переменные, объявленные с ключевым словом final, являются константами и их значение нельзя изменить после присвоения.

**Финальные методы (final methods):**

Методы, объявленные с ключевым словом final, не могут быть переопределены в подклассах.

Это применяется к методам, чтобы предотвратить изменение их реализации в подклассах.

**Финальные классы (final classes):**

Классы, объявленные с ключевым словом final, не могут иметь подклассов.

Применяется, когда необходимо предотвратить наследование.

# Перегрузка и переопределение методов. Коварианты возвращаемых типов данных

**Перегрузка методов (Method Overloading):**

Перегрузка методов — это возможность определения нескольких методов в классе с одним и тем же именем, но разными параметрами. Параметры могут отличаться по типу, количеству или порядку. Компилятор определяет, какой метод вызывать, основываясь на переданных аргументах.

**Переопределение методов (Method Overriding):**

Переопределение методов происходит, когда подкласс предоставляет реализацию метода, который уже определен в его суперклассе. Подкласс может изменить или дополнить поведение метода, но сигнатура (имя, возвращаемый тип и параметры) должна оставаться такой же.

**Коварианты возвращаемых типов данных:**

Коварианта возвращаемого типа в Java позволяет подклассам возвращать подтипы (или тот же тип) результата в переопределенных методах. Если говорить очень просто, то ковариантность позволяет нам использовать подтип (более конкретный тип) там, где ожидается супертип (более общий тип).

# Элементы функционального программирования в синтаксисе Java. Функциональные интерфейсы, лямбда-выражения. Ссылки на методы.

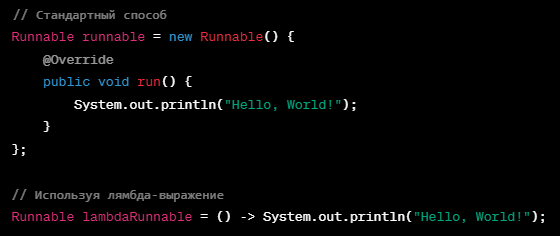
**Функциональное программирование (Functional Programming, FP)** - это парадигма программирования, основанная на использовании функций как основных строительных блоков программ. Она отличается от императивного программирования, где основной акцент делается на изменении состояния программы.

**Лямбда-выражения (Lambda Expressions):**

Лямбда-выражения представляют собой компактный способ представления анонимных функций (то есть те что не имеют явного имени). Они добавляют функциональные возможности, позволяя передавать функции как аргументы методам, возвращать их из других методов и присваивать их переменным.

Основной синтаксис лямбда-выражения выглядит следующим образом:

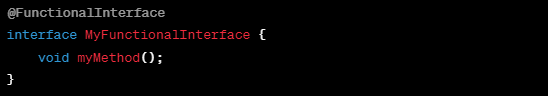
(parameters) -> expression



**Функциональные интерфейсы (Functional Interfaces):**

Предоставляет встроенные функциональные интерфейсы в пакете java.util.function, которые широко используют лямбда-выражения и ссылки на методы.

Функциональные интерфейсы - это интерфейсы, содержащие только один абстрактный метод. Они используются в контексте лямбда-выражений и других функциональных возможностей.

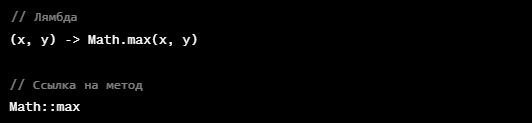


Когда вы помечаете интерфейс аннотацией @FunctionalInterface, компилятор Java проверяет, чтобы в этом интерфейсе действительно был только один абстрактный метод. Если в интерфейсе будет больше одного абстрактного метода, компилятор выдаст ошибку.

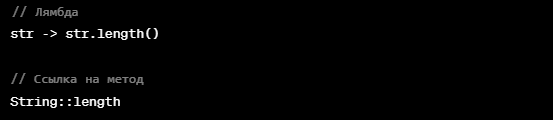
**Ссылки на методы (Method References):**

В Java ссылки на методы (Method References) представляют собой сокращенный способ передачи метода как параметра. Это удобное расширение лямбда-выражений, которое позволяет передавать методы в качестве аргументов функций или использовать их в функциональных интерфейсах.

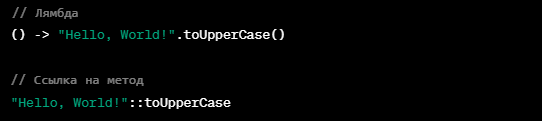
**Существует несколько видов ссылок на методы:**

**Ссылка на статический метод: **

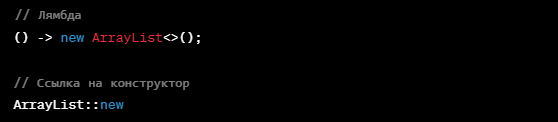
**Ссылка на метод экземпляра:**

****

**Ссылка на метод экземпляра определенного объекта:**

****

**Ссылка на конструктор:**

****