# Конспект спринта №4

## Другие модификаторы и работа с перечислениями



В Java есть группа ключевых слов, которую принято называть **другими модификаторами** (англ. *non access modifiers,* «модификаторы не-доступа»).

Они уведомляют JVM об особом поведении класса, метода или переменной.

## Модификатор `static`

Переменная, объявленная внутри класса с модификатором `static`, называется **статической**, потому что она привязана исключительно к самому классу и существует независимо от его экземпляров.

Если значение обычной переменной можно менять у каждого из объектов по отдельности, то статическая переменная является общей для всех экземпляров класса. Это значит, что:

- внутри класса существует только одна копия статической переменной;
- на её значение ссылаются все экземпляры класса;
- если изменить значение статической переменной, оно изменится у всех объектов класса.

Чтобы сделать переменную статической, при её объявлении нужно добавить слово `static`. Модификатор доступа может быть любым: `public`, `private` или `protected`.

Обращаться к статическим переменным нужно через имя класса — в формате `<имя класса>.<имя переменной>`. Ведь статическая переменная относится не к конкретному экземпляру, а к самому классу.

Переменную нужно делать статической, если:

- её значение не зависит от объектов (например, в коде будильника «Бодрое утро» всем объектам пригодится одинаковый параметр — название приложения, поэтому его можно записать в статическую переменную `static String nameOfApp = "Бодрое утро"`);
- её значение будет совместно использоваться всеми объектами одного класса (например, в статической переменной удобно хранить переключатель состояния персонажей: если в компьютерной игре применить к противникам эффект заморозки, все они должны одновременно замереть).

### Метод с модификатором `static`

`Static`-метод так же, как и `static`-переменная, принадлежит классу, а не конкретному экземпляру, и может использоваться без создания объекта. Для

того чтобы создать свой статический метод, достаточно при его объявлении добавить ключевое слово `static`:

```
public static void method() {
}
```

Внутри класса к статическому методу можно обратиться так же, как к обычному, — по имени. А для внешнего вызова можно обратиться через имя класса `<имя класса>.<имя метода>`.

Например, у класса `Integer` есть статический метод `max(int a, int b)`, который определяет наибольшее из двух переданных чисел.

Чаще  всего статические методы применяются в утилитарных (англ. *utility* — «полезный») задачах — они отвечают за выполнение полезных действий, которые не меняют состояние объекта. Например, в стандартной библиотеке Java есть класс `Arrays` (англ. «множества»). Внутри него можно найти статические методы для работы с массивами: сортировку, поиск, сравнение и другие.

**Статический метод может обращаться только к статическим переменным.**

Переопределять статические методы нельзя. Но вы можете объявить статический метод с одинаковой сигнатурой в родительском классе и классе-наследнике. Наследника можно сохранить в переменной родительского типа. Если объект-наследник сохранён в переменную родительского типа, то при выполнении программы будет вызван родительский метод, а не метод класса-наследника.

Будьте осторожны, не оставляйте таких ловушек в своём коде: не определяйте статические методы с одинаковыми сигнатурами в наследниках, лучше придумайте для них уникальные имена или объявите без модификатора `static`.

## Ключевые слова `this` и `super` запрещены в `static`-методах

Внутри статического метода нельзя использовать ключевые слова `this` и `super`. Потому что они относятся к конкретным объектам класса, а `static`-методы — к самому классу.

Чтобы такой код выполнился, необходимо добавить явное создание объекта.

### Статический импорт

К переменным и методам с модификатором `static` можно обращаться ещё одним способом — через **статический импорт** (англ. *static import*). Благодаря ему со статическими переменными и методами другого класса можно работать как с внутренними.

> 💡 Будьте аккуратны: излишне частое использование статического импорта внутри одного куска кода может сделать программу нечитаемой и неподдерживаемой. Используйте `import static`, только когда вам нужен частый доступ к статическим членам из одного или двух классов.

### Модификатор `static` перед методом `main()`

JVM выполняет код, начиная с метода `main()` — это одно из основных соглашений, принятых разработчиками. Если бы не `main()`, нам приходилось бы для каждой программы указывать метод старта.

А слово `static` необходимо для того, чтобы проект мог запускаться без объектов. Иначе нужно было бы каждую программу сопровождать пояснениями о том, какие параметры передавать в конструкторы экземпляров.

Каждый из модификаторов метода `public static void main(String[] args)` обязателен. Если не указать `static`, программа будет скомпилирована без каких-либо ошибок. Но потом, во время выполнения, JVM будет искать метод `main()` с уровнем доступа `public`, статический, с типом возвращаемого значения `void` и массивом `String` в качестве аргумента.

Если такой метод не будет найден, выполнение прервётся с ошибкой.

## Модификатор Final

Одни элементы кода могут обновлять свои значения — например, обычные и `static`-переменные, а другие должны оставаться неизменными. Для всего, что в программе менять нельзя, есть модификатор `final` (англ. «окончательный»).

### Переменная с модификатором `final`

Если при объявлении переменной добавить модификатор `final`, то после инициализации её значение станет окончательным — изменить его будет нельзя. Например:

```
public class Practicum {

    public static void main(String[] args) {
        final String figureOfEarth = "spherical"; // Инициализация final-переменной
        figureOfEarth = "flat"; // Попытка изменить значение final-переменной

        System.out.println(figureOfEarth);
    }
}
```

Мы попытались присвоить переменной `figureOfEarth` (от англ. «форма Земли») новое значение `"flat"`, и программа завершила работу с ошибкой. Потому что `figureOfEarth` — это `final`-переменная, а значит, она может быть проинициализирована только один раз.

Для переменных с примитивным типом это правило работает всегда. Если же `final`-переменная ссылается на объект, то ситуация более сложная:

- состояние объекта менять можно;
- а вот присваивать `final`-переменной другой объект нельзя.

Переменную с модификатором `final` необязательно инициализировать сразу. Это можно сделать в любой момент после её объявления и до первого применения в коде. Будьте аккуратны: любое присвоенное ей значение станет финальным.

Есть ещё одно правило. Переменная с модификатором `final` уровня класса обязательно должна быть проинициализирована:

- при объявлении — если значение для всех объектов одинаково;
- или в теле конструктора — если значение для каждого экземпляра класса уникально.

При этом и в том, и в другом случае у каждого из объектов будет своё финальное поле, а не общее, как в случае с модификатором `static`.

## Метод с модификатором `final`

Модификатор `final` защищает метод от переопределения в подклассе. Это значит, что реализация метода самодостаточна и завершена — дорабатывать или менять его в дочернем классе нельзя.

Допустим, у нас есть класс `Bicycle` (англ. «велосипед»). От него можно унаследовать подклассы для велосипедов с разными спецификациями:

- спортивных или шоссейных;
- двух-, трёх- или четырёхколёсных и т. д.

Но независимо от вида велосипеда, он обязательно должен делать две вещи:

- снижать скорость — если велосипедист нажимает на тормоз,
- и разгоняться — если активно крутятся педали.

Поэтому метод торможения `applyBrake` и метод разгона `speedUp` можно объявить с модификатором `final`, чтобы их нельзя было переопределить.

А вот к `private`-методам применять ключевое слово `final` не нужно — их и без него никогда и нигде нельзя переопределять. К конструктору тоже нет необходимости добавлять `final`, потому что он никогда не наследуется.

### Класс с модификатором `final`

Чтобы запретить наследование класса, объявите его `final`. Тогда создать от него подклассы будет невозможно.

А ещё все его методы тоже становятся `final`. Это логично: раз от класса нельзя ничего наследовать, то и переопределить его методы не получится.

> 📌 Если автор кода создал класс с модификатором `final`, значит, он хотел, чтобы его структура оставалась постоянной из соображений логики или безопасности.

Вы уже встречались с `final`-классами. Как правило, это классы-обёртки: `Integer`, `Boolean`, `Double` и другие.

## Константы

Переменная — не единственный способ хранения данных в программе. Есть ещё «постоянная», или **константа** (англ. *constant*), — она называется так, потому что изменить её значение во время работы программы невозможно.

В Java есть много констант. Вот некоторые из них:

- `MIN_VALUE` (минимальное значение) и `MAX_VALUE` (максимальное значение) класса `Integer`,
- `TRUE` и `FALSE` класса `Boolean` и многие другие.

Константа в Java — это статическое финальное поле. Чтобы его создать, примените модификаторы `static` и `final`. И обязательно инициализируйте его при объявлении. Делается это так:

```
static final тип ИМЯ_КОНСТАНТЫ = значение; // /Объявление и инициализация константы
```

Если не инициализировать `static final` константу сразу — произойдёт ошибка компиляции.

Для имён констант в Java принято использовать стиль **SCREAMING_SNAKE_CASE** (англ. «регистр кричащей змеи») — слова внутри имени пишутся в верхнем регистре и разделяются символом подчёркивания. Благодаря этому константы можно быстро отличить от обычных переменных.

Переменную с модификатором `final` тоже можно назвать константой. Она константна на уровне отдельных объектов, при этом её значение для каждого из них может быть разным.

Например, в классе `Cat` можно создать финальную переменную `final String furColor;`. В конструкторе объектов у `firstCat` ей будет присвоено значение `"grey"`, а у `secondCat` — `"white"`. И в том, и в другом случае переменная `furColor` — неизменяемая, но значения у неё при этом разные.

А константы, которые объявляются через `static final`, общие для всех объектов. У того же класса `Cat` может быть константа `static final SOUND = "МЯУ!"`, и она будет храниться в единственном экземпляре для всех объектов класса.

В константе может храниться не только единичное значение, но и полноценное выражение, включающее обращения к другим статическим полям или вызовы статических методов. Такие выражения помогают улучшить читаемость кода.

Константы помогают бороться с **магическими**, то есть не понятно что означающими числами (от англ. *magic numbers*) в коде.

## Перечисления (enum)

Кроме примитивов и классов, в Java есть специальный тип данных, который называется **перечисление** (англ. *enumerated type*, «перечисляемый тип»). Он нужен для хранения множества значений — но не любого, а ограниченного.

Для объявления перечисления применяется ключевое слово `enum`. После него пишется имя в UpperCamelCase, а затем в фигурных скобках перечисляются элементы ограниченного множества — списком, через запятую.

Все элементы перечисления принято писать как константы: в верхнем регистре, разделяя слова внутри названий символами подчёркивания. Дело в том, что перечисляемый тип по сути — это и есть список логически связанных констант.

Поэтому иногда значения перечисления так и называют: константы перечисления. Каждая из них — `static final` и не может быть изменена после создания.

Чтобы создать перечисление в IntelliJ IDEA:

- в структуре текущего проекта выберите New → Java Class;
- в появившемся окне введите имя нового файла (оно должно совпадать с названием перечисления) и выберите тип Enum.

Так же, как создаются переменные с типом `String` или `int`, можно создавать переменные с типом объявленного перечисления. В такой переменной можно, например, хранить жанр фильма.

Значение переменной `genre`, как и других переменных с типом `enum`, нужно инициализировать в упрощённом виде — без оператора `new`.

```
FilmGenre genre = FilmGenre.COMEDY;
```

Это связано с тем, что переменной с типом `enum` можно присвоить только то значение, которое определено в перечислении, а значит, существует в единственном экземпляре на всю программу. В примере с фильмами переменной `genre` можно присвоить только одно из значений перечисления `FilmGenre`.

Элементы перечисления можно сравнивать друг с другом с помощью оператора `==`.

## Метод `equals()`

Все методы класса `Object` — `toString()`, `getClass()`, `hashCode()` и другие — можно применять и к `enum`.

Метод `equals()` по назначению совпадает с оператором `==`. Дело в том, что каждая из констант перечисления хранится в единственном экземпляре. Поэтому если создать несколько переменных со значением, например, `Color.GREEN`, все они будут ссылаться на одну и ту же константу — `GREEN` (а оператор `==` как раз это и проверяет).

> 💡 Нельзя однозначно сказать, что лучше использовать: `equals()` или `==`. И у того, и у другого варианта есть свои плюсы и минусы, и среди разработчиков нет единого мнения. Сторонники `equals()` говорят о том, что любой элемент `enum` — это объект, соответственно, сравнивать его значения нужно как объекты. Сторонники `==` в свою очередь парируют, что оператор сравнения повышает читаемость.

Есть небольшой нюанс в том, как ведут себя эти методы при работе с `null`: если сравнивать объект с элементом перечисления через метод, то `equals` всегда должен быть вызван у элемента перечисления, а не у объекта, с которым мы этот элемент пытаемся сравнить. Иначе возникнет ошибка `NullPointerException`.

### Метод `values()`

Возвращает массив, содержащий все значения перечисления в том же порядке, в котором они объявлены.

Чаще всего этот метод используется в тех частях приложения, где нужно предоставить все возможные значения: в выпадающих списках, перечислениях доступных опций и так далее.

### Метод `valueOf(String name)`

Находит и возвращает константу перечисления, которая равна значению строки `name`. Если элемент не будет найден, выполнение метода завершится с ошибкой.

Такой метод будет полезен, когда одному приложению нужно принять константу перечисления от другого приложения.

### Метод `name()`

Возвращает имя элемента перечисления. На первый взгляд может показаться, что методы `name()` и `toString()` дают одинаковый результат, но это не так.

Разница вот в чём:

- метод `name()` объявлен с модификатором `final` — его нельзя переопределять, но можно уверенно использовать для получения оригинального имени элемента перечисления;

- а метод `toString()` может быть переопределён — с его помощью можно вернуть адаптированное и более понятное для пользователя имя константы.

Если рядом с набором констант есть метод, то после последнего элемента перечисления нужно поставить символ `;`.

### Перечисления и оператор `switch`

Обычно каждый элемент перечисления требует особой обработки. Удобнее всего делать это с помощью оператора выбора `switch`.

Если в перечислении всего три элемента, их можно обработать и через обычную конструкцию `if-else`. Но в случаях, когда в перечислениях находятся десятки или даже сотни значений, оператор `switch` незаменим. Он делает код более понятным и читаемым.

# Абстракция и полиморфизм

# Абстракция

**Абстракция** (англ. *abstraction,* «отвлечение») как принцип ООП — это сокрытие деталей реализации: у нас есть информация о том, ***что*** делает объект, но не о том, ***как*** он это делает.

Например, для работы с объектами класса `ArrayList` вам не нужна информация о том, что они хранят элементы в обычном массиве и создают новый, если в старом заканчивается свободное место. Вам достаточно знать, что они умеют хранить элементы и добавлять новые.

Абстрактный класс — это базовый класс, у которого не может быть экземпляров. На его основе создаются обычные классы, объединённые общими чертами.

Создавать объекты, относящиеся к абстрактным классам, можно в классах-наследниках. Об этом мы расскажем дальше.

Методы в абстрактных классах могут быть двух видов:

- обычные (они пишутся с реализацией, которая будет общей для всех классов-наследников);
- и абстрактные (они указываются без реализации, потому что у каждого из классов-наследников она будет своя).

## Обычные методы в абстрактном классе

Если у группы разных объектов реализация какого-то действия совпадает, то её лучше написать сразу в абстрактном классе, внутри обычного метода. В таком случае не придётся дублировать один и тот же код во множестве классов.

Например, и лягушки, и жабы одинаковым образом реализуют метод `eat()` — и те, и другие с удовольствием едят насекомых. Поэтому можно сделать его общим. Для этого внутри абстрактного класса объявим обычный метод `eat()` и напишем его реализацию.

А чтобы объявить абстрактный метод, перед типом возвращаемого значения необходимо указать ключевое слово `abstract`. Тело метода при этом будет отсутствовать — вместо него ставится точка с запятой. Абстрактные методы дают информацию только о том, что сможет делать объект класса-наследника. Например, передвигаться по суше — `move()`.

```
public abstract class Amphibian {

    public void eat() { // Обычный метод с реализацией
        System.out.println("Кушаю насекомых!");
    }

    public abstract void move(); // Абстрактный метод без реализации
}
```

Абстрактный класс, в котором есть только обычные методы, всё равно будет абстрактным. А вот если в обычном классе появится хотя бы один абстрактный метод — нужно будет этот класс объявить абстрактным, иначе возникнет ошибка.

## Наследники абстрактного класса

Абстрактный класс — это только заготовка, которая становится конкретной и реализуется в полной мере только в классах-наследниках.

Класс-наследник должен реализовать все унаследованные абстрактные методы, иначе при компиляции программы возникнет ошибка: `<Class name> is not abstract and does not override abstract method <method name> in <abstract class name>` (англ. `[Класс] не является абстрактным и не переопределяет метод из [абстрактного класса]`).

Допустим, разработчик не хочет реализовывать в классе-наследнике все абстрактные методы базового класса. Тогда он обязательно должен объявить класс-наследник также абстрактным.

## Объекты классов-наследников

У абстрактного класса не может быть объектов. Зато они могут быть у его классов-наследников. И объявляются эти объекты через конструктор конкретного класса.

При создании объектов в программе будет вызван конструктор конкретного класса. Абстрактный класс содержит конструктор по умолчанию, но вы можете определить вместо него любые конструкторы с параметрами.

## Интерфейс

Интерфейс — это совокупность методов без реализации, которые описывают некоторый функционал. В дальнейшем он может быть имплементирован, или реализован (от англ. *implement*, «реализовывать») в его классах-реализациях. На основе интерфейса нельзя создавать объект: у него нет конструктора по умолчанию и в него нельзя добавить конструкторы с параметрами. В чём-то интерфейсы похожи на абстрактные классы, но всё же между ними есть различия. И самое главное — концептуальное:

- Абстрактные классы нужны для того, чтобы у всех классов-наследников создавать и поддерживать общую *структуру*. Они как бы говорят: «Все мои наследники будут похожи на меня: и свойствами, и методами!».
- Интерфейсы нужны для добавления в класс-реализацию определённой *функциональности.* Их девиз мог бы быть таким: «Объекты класса, который имплементирует меня, научатся делать кое-что определённое!».

Объявление интерфейса похоже на объявление класса, только вместо `class` используется ключевое слово `interface`. Внутри указываются методы без реализации: все методы интерфейса являются абстрактными по умолчанию. Модификаторы доступа писать не нужно — все методы интерфейса по умолчанию являются публичными.

Чтобы класс реализовывал интерфейс, необходимо после названия класса указать ключевое слово `implements` и имя интерфейса, а над реализацией метода интерфейса указать аннотацию `@Override`:

```
import java.util.ArrayList;
import java.util.List;

public class CalendarApp implements NoteBook {
    List<String> notes = new ArrayList<>();

    @Override
    public void addNote(String note) {
        notes.add(note);
        System.out.println("Заметка успешно добавлена!");
    }
}
```

Класс обязательно должен либо реализовать все методы интерфейса, либо объявить себя абстрактным — иначе при компиляции возникнет ошибка: `<Class name> is not abstract and does not override abstract method <method name> in <interface name>` (англ. `[Класс] не является абстрактным и не переопределяет метод из [интерфейса]`).

В интерфейсе можно объявить переменные, но они всегда будут константами. Поэтому в переменных интерфейса часто сохраняют значения, которые нужно использовать в различных частях программы — их модификатор по умолчанию тоже будет `public`, как и у методов.

Механизм наследования в Java очень удобен, но у него есть важное ограничение — **наследоваться можно только от одного класса**. Один класс может реализовывать сразу несколько интерфейсов.

# Полиморфизм

**Полиморфизм** (в переводе с греческого означает «многообразный») — это способность принимать разные формы. Один из ярких примеров полиморфизма в химии — модификации углерода. Он может принимать форму графита (и тогда его вставляют в карандаши) или форму алмаза (и тогда он отправляется на огранку к ювелиру).

В применении к языкам программирования полиморфизм означает способность программы одинаково работать с объектами, если они имеют одинаковый интерфейс. При этом код может ничего не знать о конкретном типе этого объекта.

Полиморфизм в программировании проявляется не только в особенностях работы с интерфейсами, но и в механизме наследования. По аналогии с интерфейсами можно сохранять объект наследника в переменную с родительским типом. При наследовании в Java классы связываются **отношением IS-A** (англ. «является»): один класс является подклассом другого. Все капибары — грызуны, но не все грызуны — капибары.

**В Java полиморфизм реализуется и другими способами. Выделяют несколько видов:**

- Классический полиморфизм;
- Ad-hoc полиморфизм, который делится на два подвида:
  - Динамический,
  - Статический;
- Параметрический полиморфизм.

## Классический полиморфизм

Если разные классы имплементируют одинаковый интерфейс или наследуются от одного класса, их объекты будут вести себя одинаково. Это позволяет программе работать с ними одинаковым образом — независимо от их типа.

## Динамический ad-hoc полиморфизм

Кроме классического, в Java реализован ещё один тип полиморфизма — **Ad-hoc-полиморфизм** («специальный полиморфизм»). Динамический полиморфизм тесно связан с наследованием и заключается в переопределении (англ. *overriding*) методов — он позволяет им демонстрировать различное поведение при вызове для разных типов. Если в программе есть переменная с типом родителя и в ней хранится объект класса-наследника, то при вызове метода, который определён и там, и там, будет вызван метод класса-наследника. Динамический полиморфизм используется, чтобы поменять поведение класса, не меняя его код.

## Статический ad-hoc полиморфизм

**Статический ad-hoc полиморфизм** — это **перегрузка** (англ. *overloading*) **метода**. В классе может быть несколько методов с одинаковым названием, но с разными типами параметров. В этом случае компилятор сам выберет нужную реализацию — в зависимости от типа переданного аргумента, который становится известен на этапе компиляции.

Статический полиморфизм часто встречается в стандартной библиотеке Java — он позволяет расширить интерфейс класса через перегрузку метода. Если есть метод, который часто вызывается с одинаковым значением параметра, то можно перегрузить его, зафиксировав это значение внутри метода. Так его станет проще вызывать.

## Параметрический полиморфизм (Дженерики)

**Параметрический полиморфизм** — вид полиформизма, позволяющий реализовать поведение класса или метода с указанием параметров и возвращаемых значений общего вида вместо конкретных. Дженерики позволяют писать классы и алгоритмы, поведение которых не зависит от конкретного типа. При этом вы можете указать, какие типы будут использоваться определённым классом, а Java сама будет проверять на этапе компиляции программы, что программист не допустил ошибку и не передал значения неверного типа.  Java позволяет добавить ограничение на верхнюю границу дженерика (англ. *generic upper bound*). Это означает, что на месте

дженерика можно будет использовать только те типы, которые наследуются от типа, указанного в виде верхней границы.

Объявить класс с дженериком можно следующим образом:

```
class GeneralClass<T> {
    T element; // Тип поля element будет таким, который мы укажем при создании объекта класса GeneralClass
}

GeneralClass<Double> doubleObj = new GeneralClass<>(); // Можно опустить тип во вторых скобках
doubleObj.element = 100.0;
```

При вызове дженерика все примитивные типы нужно заменять обёртками.

Дженерики можно использовать не только с классами, но с интерфейсами. Для этого после имени интерфейса в угловых скобках необходимо указать параметр типа `T` (или несколько параметров через запятую, если интерфейс должен использовать несколько дженериков).