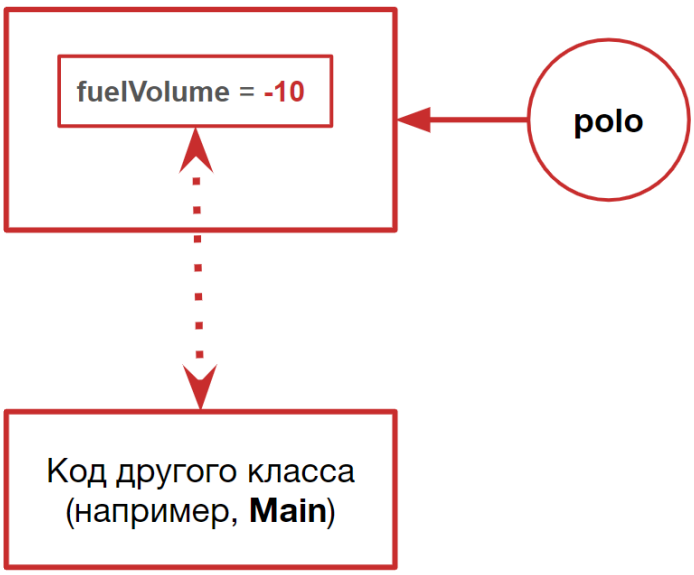
Инкапсуляция

Проблема открытости состояния

```
class Car {
    double fuelVolume = 0; // объем топлива
    // …

}

Car polo = new Car(45, 6);
polo.fuelVolume = -10;
```

В текущей реализации класса **`Car`**, внутреннее состояние объектов этого класса, такое как объем топлива (**`fuelVolume`**), является открытым и доступным для прямого доступа извне, что не является хорошей практикой в объектно-ориентированном программировании. Такой подход позволяет коду, находящемуся вне класса **`Car`**, например, в классе **`Main`**, не только читать, но и изменять значения полей объектов Car напрямую, что может привести к некорректному или неожиданному состоянию объекта. Например, в данном случае, объем топлива может быть установлен в отрицательное значение (**`polo.fuelVolume = -10;`**), что не имеет физического смысла и указывает на потенциальную ошибку в логике программы.

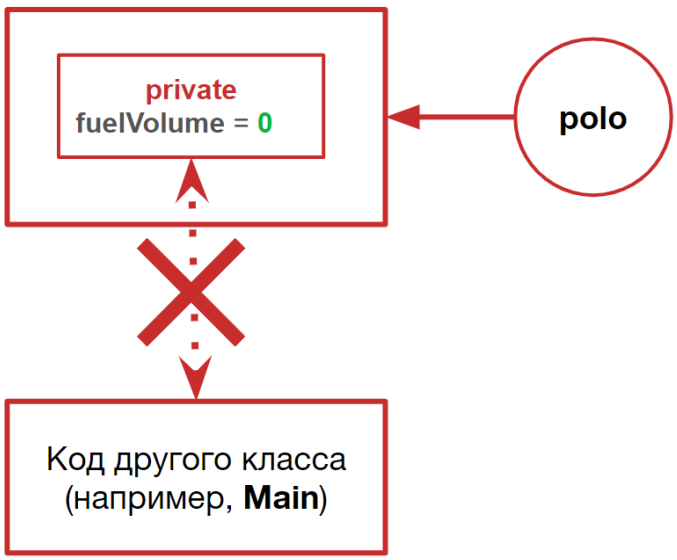


Приватные поля

Использование модификатора доступа **private** является фундаментальным механизмом для защиты внутреннего состояния объектов. Применение `private` к полям класса ограничивает доступ к этим полям, делая их недоступными для чтения или изменения напрямую из кода за пределами класса. Такой подход позволяет разработчикам контролировать взаимодействие с внутренними данными объекта.

```
class Car {
    private double fuelVolume = 0;

    // ...

}

Car polo = new Car(45, 6);

// ошибка компиляции
polo.fuelVolume = -10;
```

В примере с классом `Car`, объявление поля `fuelVolume` с модификатором `private` означает, что попытка прямого доступа к этому полю извне класса, например, для его изменения (`polo.fuelVolume = -10;`), приведет к ошибке компиляции. Таким образом, внутреннее состояние объекта `Car` защищено от непреднамеренных или некорректных изменений, обеспечивая целостность и корректность работы программы.



Приватные методы

Модификатор доступа private играет важную роль не только в защите полей класса, но и в ограничении доступа к его методам, делая их недоступными для вызова вне класса. Это позволяет "спрятать" вспомогательные или внутренние методы, которые предназначены для использования только в рамках самого класса. Такой подход способствует модульности кода, позволяя скрыть детали реализации и выделить публичный интерфейс класса для взаимодействия с внешним кодом.

```
class Car {
    private double fuelVolume = 0;

    // …

    boolean go(double kilometers) {
        double fuelNeeded =
                    calcFuel(kilometers);

        // …
    }

    // метод для расчета необходимого топлива
    private double calcFuel(double kilometers) {
        return (kilometers * fuelConsumption) / 100;
    }

}

Car polo = new Car(45, 6);

// ошибка компиляции
polo.calcFuel(5);
```

В примере класса `Car`, метод `calcFuel`, помеченный как `private`, используется для расчета необходимого количества топлива для поездки на заданное количество километров. Этот метод является вспомогательным и предназначен для внутреннего использования в классе, например, в методе `go`, который является публичным и предоставляет функциональность по перемещению автомобиля на определенное расстояние. Применение модификатора `private` к методу `calcFuel` гарантирует, что этот метод не может быть вызван напрямую из объектов класса `Car`, созданных во внешнем коде, как показано в примере с попыткой вызова `polo.calcFuel(5)`, что приведет к ошибке компиляции.

Использование **`private`** для методов улучшает структурирование кода, позволяя четко разделить внутреннюю логику работы класса на отдельные, легко управляемые части, и обеспечивает дополнительный уровень защиты от неправильного использования класса. Также это способствует сокрытию сложности и деталей реализации от пользователя класса, предоставляя ему только необходимый и безопасный для использования интерфейс.

Getters

Использование модификатора **`private`** для полей предотвращает несанкционированное изменение или чтение данных объекта, что может привести к некорректному поведению программы. Однако, для обеспечения возможности чтения этих скрытых данных, без предоставления возможности их изменения, применяются специальные методы доступа, известные как геттеры (getters).

```
class Car {
    private double fuelVolume = 0;

    // …

    double getFuelVolume() {
        return fuelVolume;
    }

}

Car polo = new Car(45, 6);
polo.refuel(30);

// теперь можем получить значение
System.out.println(polo.getFuelVolume());
```

В примере с классом **`Car`**, поле **`fuelVolume`**, хранящее информацию об объеме топлива, объявлено как **`private`**, что делает его недоступным для прямого доступа вне класса. Для того чтобы предоставить безопасный доступ к значению этого поля, в классе реализован метод **`getFuelVolume`**. Этот метод не принимает никаких параметров и возвращает текущее значение поля **`fuelVolume`**, позволяя тем самым получать информацию о состоянии объекта.

Setters

Так же, как и методы для чтения данных (геттеры), используются методы для их изменения, называемые сеттерами (setters). Эти методы позволяют обновлять значения полей объекта, при этом предоставляя возможность для выполнения дополнительных

проверок корректности предоставляемых значений, тем самым обеспечивая целостность и корректность состояния объекта.

```
class Car {
    private double fuelVolume = 0;

    // …

    void setFuelVolume(double fuelVolume) {
        if (fuelVolume <= 0) {
            System.err.println(“Некорректное  значение”);
        } else {
            this.fuelVolume = fiuelVolume;
        }
    }
}

Car polo = new Car(45, 6);
polo.refuel(30);

// сообщение об ошибке
polo.setFuelVolume(-10);

// установка значения
polo.setFuelVoume(10);
```

В примере с классом `Car`, реализован метод `setFuelVolume`, который служит для обновления значения поля `fuelVolume`, отвечающего за объем топлива в автомобиле. Важной особенностью сеттера является включение проверки корректности предоставляемого значения: если значение параметра `fuelVolume` меньше или равно нулю, метод выводит сообщение об ошибке, указывая на некорректность ввода. В случае, если предоставленное значение является корректным (больше нуля), сеттер обновляет значение поля `fuelVolume`, используя ключевое слово `this` для различения между параметром метода и полем объекта.

Использование сеттера `setFuelVolume` в коде представлено двумя примерами: сначала производится попытка установить значение поля `fuelVolume` равным `-10`, что приводит к выводу сообщения об ошибке, так как значение является некорректным. Затем значение поля успешно обновляется на `10`, поскольку это значение удовлетворяет условиям проверки в сеттере.

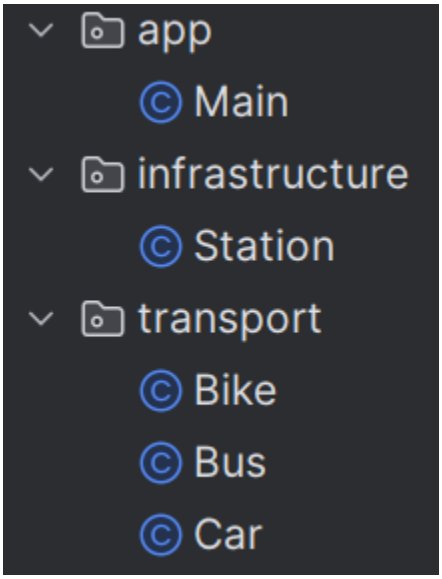
Применение сеттеров с проверками корректности значений позволяет не только предотвратить установку объекта в некорректное или нелогичное состояние, но и упрощает отладку и поддержку программы, явно указывая на ошибки в данных или логике их обработки.

Пакеты, публичный доступ

Пакеты используются для организации кода и его структурирования путем группировки связанных классов. Это не только помогает в управлении пространством имен, но и улучшает модульность, переиспользование и обеспечение безопасности кода. Пакеты позволяют разрабатывать более четкую структуру приложения, упрощая его поддержку и развитие.

Когда модификатор доступа не указан для поля, метода или конструктора, доступ к этим элементам ограничен контекстом текущего пакета. Это означает, что такие поля, методы и конструкторы не доступны для классов, расположенных в других пакетах. Например, если в пакете **`transport`** определен класс **`Car`** с методом **`refuel()`** без модификатора доступа, классы из пакетов **`app`** и **`infrastructure`**, такие как **`Main`** и **`Station`**, не смогут вызвать этот метод.

В то же время, все классы внутри одного пакета, например **`transport`**, могут свободно взаимодействовать друг с другом, имея доступ к "пакетным" полям, методам и конструкторам без каких-либо ограничений. Это способствует удобной организации связанных классов, обеспечивая их взаимодействие в рамках одного модуля или компонента системы.



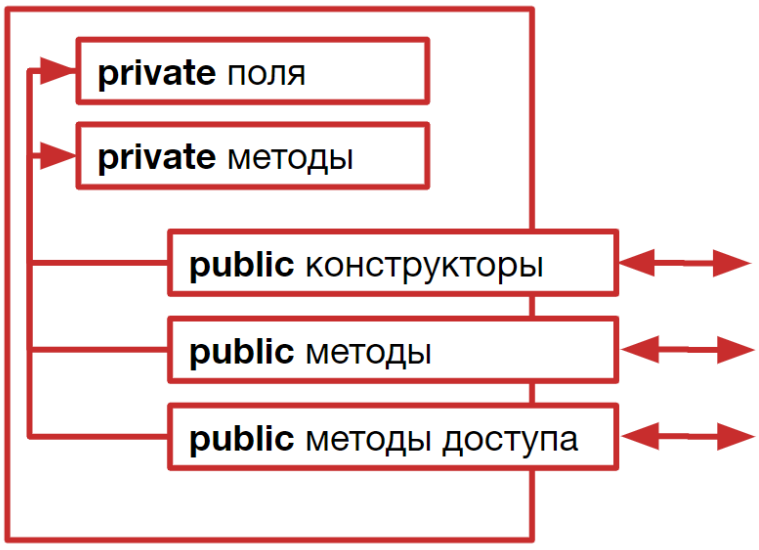
Для того чтобы сделать элемент класса доступным в любом месте программы, независимо от пакета, к которому принадлежит класс, используется модификатор доступа **public**. Определение элемента класса как **public** является явным указанием на то, что данный элемент предназначен для широкого использования и может быть безопасно вызван из любого другого класса вне зависимости от его пакетной принадлежности.

Модификаторы и уровни доступа

МОДИФИКАТОР	КЛАСС	ПАКЕТ	ДЛЯ ВСЕХ
public	Да	Да	Да
отсутствует	Да	Да	Нет
private	Да	Нет	Нет

Инкапсуляция

Инкапсуляция является одним из основных принципов объектно-ориентированного программирования (ООП), обеспечивающим сокрытие внутренней реализации класса от внешнего мира, а также интеграцию данных (состояния) и кода (поведения), которые манипулируют этими данными, в единую структурную единицу — класс. Этот принцип направлен на упаковку данных и методов, работающих с данными, внутри класса и на ограничение доступа к ним из других частей программы для предотвращения нежелательного вмешательства или ошибочного использования.



Основными целями инкапсуляции являются:

Сокрытие внутренней реализации класса: Инкапсуляция позволяет скрыть детали реализации класса, предоставляя внешнему коду только необходимый интерфейс для взаимодействия с объектом. Это означает, что изменения внутренней реализации класса

не будут иметь прямого влияния на код, который использует этот класс, что упрощает модификацию и развитие программы.

Защита данных: Инкапсуляция защищает состояние объекта от прямого доступа и модификации извне, позволяя изменять данные объекта только через его методы. Это помогает предотвратить неправомерное использование объектов и обеспечивает корректность данных, так как доступ к изменению полей объекта может быть строго контролируем через методы, которые могут включать проверки валидности и корректности данных.

В Java инкапсуляция достигается за счет использования модификаторов доступа (например, **private**, **public** и **protected**) для полей, методов и конструкторов. Поля класса обычно объявляются как private, что ограничивает их прямой доступ из других классов, а доступ к этим полям обеспечивается через публичные методы — геттеры (для чтения данных) и сеттеры (для изменения данных). Это позволяет осуществлять контролируемый доступ к внутреннему состоянию объекта, а также предоставлять к нему ограниченный интерфейс для взаимодействия.