нетология

Модификаторы доступа, наследование

Юрий
Пеньков

Юрий Пеньков

Java Software Engineer в InnoSTage

Юрий Пеньков

План занятия

1. Модификаторы доступа
2. Наследование полей
3. Наследование методов
4. Конструкторы при наследовании

Модификаторы доступа

Инкапсуляция

Рассмотрим пример:

```
public class Book {
    // Название книги
    private String title;
    // Автор книги
    private String author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

Инкапсуляция

Вы наверняка обратили внимание на выражение внутри нашего метода:

```
this.title = title
```

Объяснение этой конструкции очень простое.

Внутри нашего класса уже есть свойство title (private String title).

И мы хотим именно свойству присвоить значение, которое находится в формальном параметре title.

Инкапсуляция

Но простое написание

```
title = title
```

привело бы нас к бесполезной операции — мы присвоили бы переменной (формальному параметру) title ее же значение.

Наша цель — присвоить значение из локальной переменной (фактического параметра) в свойство объекта. И именно для этого используется ключевое слово this. Это просто ссылка на сам объект внутри его метода.

Казалось бы, почему бы не объявить все переменные и методы с модификатором public, чтобы они были доступны в любой точке программы вне зависимости от пакета или класса?

Инкапсуляция

Возьмем, например, поле title, которое представляет название книги. Если другой класс имеет прямой доступ к этому полю, то есть вероятность, что в процессе работы программы ему будет передано некорректное значение, например, значение в некорректной кодировке. Подобное изменение данных не является желательным.

Либо же мы хотим, чтобы некоторые данные были доступны напрямую, чтобы их можно было вывести на консоль или просто узнать их значение. В этой связи рекомендуется как можно больше ограничивать доступ к данным, чтобы защитить их от нежелательного доступа извне (как для получения значения, так и для его изменения).

Использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом. Подобное сокрытие данных внутри некоторой области видимости называется **инкапсуляцией**.

Вспоминаем прошлые занятия

Модификаторы доступа

В Java используются следующие модификаторы доступа:

- `public`: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.
- `private`: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.
- `protected`: такой класс или член класса доступен из любого места в текущем классе или пакете или в классах-наследниках, даже если они находятся в других пакетах
- `default:` модификатор по умолчанию. Когда мы не пишем модификатора доступа, он по умолчанию имеет значение default. Такие поля или методы видны всем классам в текущем пакете.

Модификаторы доступа

Модификаторы доступа пишутся перед названиями переменных, методов и даже классов. А теперь определение, которое, как мне кажется, наиболее точно определяет суть инкапсуляции:

Переменные состояния объекта скрыты от внешнего мира. Изменение состояния объекта (его переменных) возможно ТОЛЬКО с помощью его методов (операций).

Это существенно ограничивает возможность введения объекта в недопустимое состояние и/или несанкционированное разрушение этого объекта. Особенно это заметно при применении многопоточности, либо использовании каких-то счетчиков.

Если влиять на переменную напрямую считается плохой практикой, то как по-другому, правильно можно изменять переменные?

Модификаторы доступа

Для этого существуют специальные методы — так называемые Геттеры и Сеттеры. Ну, они не то чтобы специальные — просто настолько часто используются, что были вынесены в отдельную категорию методов.

Геттер — от англ. «get», «получать» — это метод, с помощью которого мы получаем значение переменной, т.е. ее читаем.

```
public String getAuthor() {
        return author;
    }
```

Сеттер — от англ. «set», «устанавливать» — это метод, с помощью которого мы меняем, или задаем значение переменной.

```
public void setAuthor(String author) {
      this.author = author;
   }
```

Если результат такой же, зачем это все было менять?

Модификаторы доступа

Тут у нас в каждом методе всего по одной строчке, но если нам понадобится добавить какую-то логику — например, присваивать новое значение строке author только если она больше какой-то длины, или необходимо проверить действительно ли человек с данной фамилией является автором книги.

Главное, мы получаем контроль над происходящим — никто не может просто так менять или читать наши переменные.

Наследование полей

Пример, имеется класс Book описывающий отдельную книгу:

```
public class Book {
    // Название книги
    private String title;
    // Автор книги
    private String author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public void display(){

        System.out.println("Title: " + title + ", author: " + author);
    }
}
```

Наследование полей

И необходимо добавить класс, описывающий книгу для специалистов — класс `ProfessionalBook`.

Так как этот класс реализует тот же функционал, что и класс `Book`, так как книга для специалистов — это также и обычная книга, имеющая автора и название, то было бы рационально сделать класс `ProfessionalBook` производным (наследником, подклассом) от класса `Book`, который, в свою очередь, называется базовым классом, родителем или суперклассом:

```
class ProfessionalBook extends Book {
}
```

Наследование полей

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово `extends`, после которого идет имя базового класса. Для класса `ProfessionalBook` базовым является `Book`, и поэтому класс `ProfessionalBook` наследует все те же поля и методы, которые есть в классе Book.

Наследование (англ. inheritance) — это механизм, позволяющий создавать классы на основе другого класса.

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Так, один класс может «наследовать» характеристики другого — его методы и переменные.

Наследование полей

- Класс, являющийся основой, называют: базовым, супер, родительским.
- Класс, который создают, называют: потомок, наследник или производный класс.

Естественно, сама Java тоже использует механизм наследования. Например, все классы языка наследуют класс `Object`.

Наследование методов

Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором `private`. При этом производный класс также может добавлять свои поля и методы:

```
class ProfessionalBook extends Book {
    // Название книги
    private String specialization;

    @Override
    public void display(){

        System.out.println("Title: " + title + ", specialization: " +   specialization);
    }
}
```

Наследование методов

Перед переопределенным методом указывается аннотация `@Override`. Данная аннотация в принципе не обязательна.

При переопределении метода он должен иметь уровень доступа не меньше, чем уровень доступа в базовом класса.

Например, если в базовом классе метод имеет модификатор `public`, то и в производном классе метод должен иметь модификатор `public`.

Правила наследования

- Правило 1. Наследоваться можно только от одного класса.

Java не поддерживает наследование нескольких классов. Один класс — один родитель.

Обратите внимание — нельзя наследовать самого себя!

- Правило 2. Наследуется все, кроме приватных переменных и методов.

Все методы и переменные, помеченные модификатором private, недоступны классу-наследнику.

- Правило 3. Переделать метод класса-родителя.

Представим, что мы наследуем класс, но нам не подходит, что мы унаследовали. Допустим мы хотим, чтобы определенный метод работал не так, как в родителе. Для того, чтобы переопределить метод класса-родителя, пишем над ним `@Override`.

Правила наследования

- Правило 4. Вызываем методы родителя через ключевое слово super.

Представим, что Вы хотите изменить метод родительского класса совсем чуть-чуть — буквально дописать пару строк. Тогда в своем методе мы можем вызвать родительский метод с помощью ключевого слова `super`.

```
class ProfessionalBook extends Book {
    // Название книги
    private String specialization;

    @Override
    public void display(){
        super.display();
        System.out.println("Title: " + title + ", specialization: " +   specialization);
    }
}
```

Правила наследования

- Правило 5. Запрещаем наследование.

Если Вы не хотите, чтобы кто-то наследовал Ваш класс, поставьте перед ним модификатор final. Например:

```
final class Book {
}
```

Теперь попробуем создать наследника:

```
class FantasticBook extends Book {
}
```

Получим ошибку.

Конструкторы при наследовании

Если в базовом классе определены конструкторы, то в конструкторе производного классы необходимо вызвать один из конструкторов базового класса с помощью ключевого слова super.

Например, класс Book имеет конструктор, который принимает два параметра. Поэтому в классе `ProfessionalBook` в конструкторе нужно вызвать конструктор класса `Book`. После слова `super` в скобках идет перечисление передаваемых аргументов.

Таким образом, установка названия профессиональной книги делегируется конструктору базового класса.

При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса.

Конструкторы при наследовании

```
public class Book {
    private String title;
    private String author;

    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

}
```

```
class ProfessionalBook extends Book {
    private String specialization;

    ProfessionalBook(String specialization, String title, String author) {
        super(title, author);
        this.specialization = specialization;
        // ...
    }
}
```

Конструкторы при наследовании

Вызов конструктора суперкласса должен быть первой инструкцией в конструкторе дочернего класса. Можно вызвать конструктор суперкласса без параметров (конструктор по умолчанию):

```
class ProfessionalBook extends Book {
    private String specialization;

    ProfessionalBook(String specialization, String title, String author) {
        super();
        this.specialization = specialization;
        // ...
    }
}
```

Пример текста на слайде

Если вы не вставили ни одного явного вызова конструктора родительского класса, то компилятор Java автоматически добавит вызов конструктора родительского класса без параметров (конструктора по умолчанию).

Если конструктор родительского класса без параметров недоступен из-за модификатора доступа, или конструктора без параметров нет в родительском классе, то возникнет ошибка компиляции.

При создании экземпляра любого объекта происходит цепочка вызовов конструкторов от конструктора создаваемого объекта до конструктора класса Object. Это называется цепочкой вызова конструкторов (constructor chaining).

Чему мы научились

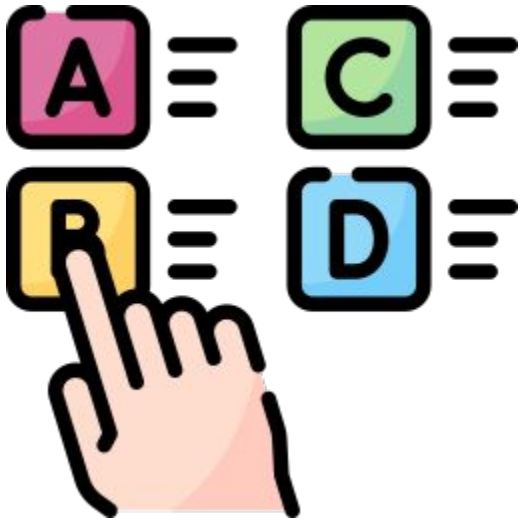
- Узнали, как использовать инкапсуляцию и модификаторы доступа;
- Узнали, как использовать наследование;
- Узнали как наследуются поля, методы и конструкторы.

Домашнее задание

Закрепите тему сегодняшней лекции — пройдите **квиз**!

В квизе вас ждут:

- пояснения к каждому варианту ответа,
- неограниченное количество попыток.



нетология

Задавайте вопросы и пишите отзыв о лекции!

Юрий Пеньков

Юрий Пеньков