



Урок № 36



Курс: «Разработка программного обеспечения на Java»

Тема: Объектно-ориентированное программирование

План

1. Введение в объектно-ориентированное программирование
2. Понятие класса и объекта
3. Конструкторы
4. Поля и методы. Ключевое слово `this`
5. Перегрузка методов и конструкторов
6. Наследование
7. Понятие интерфейса
8. Вложенные и внутренние классы
9. Анонимные классы
10. Шаблоны (Generics)

1. Введение в объектно-ориентированное программирование

Основная парадигма объектно-ориентированного программирования – наличие трех основных понятий: инкапсуляция, наследование и полиморфизм.

Все эти свойства применимы к объектам, ведь ООП – это и есть способ организации программы путем группировки кода в виде объектов. Объекты описываются отдельными полями (свойствами) и поведением (методам).

Инкапсуляция программных компонентов – скрывание прямого доступа из вне. Благодаря инкапсуляции можно ограничить нежелательный доступ к членам класса или защитить их от неправильного использования для тех частей программы, которые существуют вне объекта. Для скрывания компонентов класса используйте модификатор доступа `private`.

Концепция наследования позволяет создавать новые классы на базе существующих. Такие классы могут расширять характеристики базового класса или переопределять существующие. Наследование помогает значительно сократить однотипный программный код. Подробнее о наследовании будет описано в следующих пунктах.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого класса действий. Это значит, что объекты классов наследников можно рассматривать как объект класса родителя при определенных условиях, а именно, при работе с общими свойствами и методами, которые заложены в базовый интерфейс. Если упростить понимание понятия полиморфизма, необходимо его перевести: «одно имя, но много форм».

Рассмотри пример, который демонстрирует возможности полиморфизма. У ряда классов, которые наследуют от класса `Shape`, есть метод `draw()` (как и у класса родителя). Каждый конкретный класс переопределяет этот метод для отображения фигуры со своими уникальными характеристиками. Полиморфизм позволяет сказать, что все указанные фигуры являются экземплярами класса `Shape` и вызвать у них метод `draw()`. При этом, в зависимости от того, какая реализация этого метода в конкретном классе, будет отображена та или иная фигура.

2. Понятие класса и объекта

Класс и объект – это два ключевых понятия в ООП. С помощью класса описывается некоторая сущность. Для описания любого предмета достаточно указать его характеристики (размер, цвет, скорость и т.п.) и поведение (движение, расчеты, визуализация и т.д.). Объектом же является экземпляр этого класса (конкретный представитель), который имеет свои уникальные значения заданных в классе характеристик.

Пример:

При разработке гоночной игры, используется класс Car. Машина имеет конкретный набор характеристик: модель, цвет, скорость и т.д.. В процессе игры пользователь меняет автомобили (объекты), увеличивая их скорость и другие характеристики. Получается, что при изменении объекта, модернизируются поля экземпляра класса, но его структура остается неизменной.

Дополнительная информация:

- Все классы в Java наследуются от класса Object.
- Доступ к полю или методу объекта осуществляется через . (оператор точка).
- Если объект объявить, но не проинициализировать, тогда он будет равен null (пустая ссылка). Если обратиться к методам или полям такого объекта, произойдет ошибка.

3. Конструкторы

Конструктор предназначен для инициализации всех компонентов класса, при создании нового объекта этого класса. Имя конструктора всегда совпадает с именем класса, в котором он объявлен. Конструктор не имеет тип возвращаемого значения.

Конструктор вызывается командой `new имя_класса(<аргументы>);`

Пример: `MyObject obj = new MyObject();`

Конструктором по умолчанию называется конструктор без параметров, который неявно определен в каждом классе. Для переопределения логики конструктора по умолчанию, опишите в классе конструктор с пустыми скобками.

Пример:

```
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
    }  
}  
  
class A {  
    A(){  
        System.out.println("Default constructor");  
    }  
}
```

Как видно из примера, при создании объекта вызовется конструктор класса A и произойдет вывод сообщения в консоль.

4. Поля и методы. Ключевое слово **this**

Поля класса могут быть как примитивного типа, так и объектами другого класса. Для определения полей констант, используйте ключевое слово `final`. По соглашению имен, константы в Java необходимо записывать заглавными буквами, каждое отдельное слово пишется через нижнее подчеркивание (пример: `CONSTANT_FIELD`).

Следующий участок кода вызовет ошибку, т.к. нельзя изменять значение константы:

```
public class Main {  
  
    private static final int DEFAULT_INT = 1;  
  
    public static void main(String[] args) {  
        DEFAULT_INT = 5;  
    }  
}
```

При определении констант в полях класса, принято добавлять ключевое слово `static`, т.к. значения этого поля изменить невозможно, а если этого не делать, каждый отдельный экземпляр класса будет хранить в памяти отдельную версию этой константы.

Поля, как и методы, могут быть статическими. Для этого, используйте ключевое слово `static`. Статические поля, которые объявлены в классе, являются общими для всех объектов этого класса, поэтому, при изменении этого поля в объекте А, оно изменится и в объекте В.

Рассмотрим пример:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(A.staticField);  
  
        A a = new A();  
        A b = new A();  
  
        System.out.println(a.staticField);  
        a.staticField = 5;  
        System.out.println(b.staticField);  
    }  
}
```

```
class A {  
    public static int staticField = 1;  
}
```

Отсюда следует, что лучше обращаться к статическому члену через имя класса, а не имя объекта.

Может потребоваться, чтобы метод ссылался на вызвавший его объект. Для этого в Java определено ключевое слово `this`. Оно может использоваться внутри любого метода для ссылки на текущий объект. Также, в Java существуют понятия `getter` и `setter`. Эти методы, имя которых начинается с приставки `get` и `set` соответственно, предназначены для получения и записи значений в переменную. Применяв инкапсуляцию, мы можем скрыть переменную, но предоставит пользователю возможность получить это значение. В языке Java принято определять `get` и `set` методы для полей моделей или главных полей любого класса, а само поле делать `private`.

Рассмотрим пример:

```
public class Main {  
    public static void main(String[] args) {  
        File file = new File();  
        file.setName("My file");  
        file.setData(String.valueOf(new Random().nextInt()));  
        File fileCopy = file.copy();  
        System.out.println("New file name: " + fileCopy.getName());  
        System.out.println("New file data: " + fileCopy.getData());  
    }  
}
```

```
class File {  
    private String name;  
    private String data;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getData() {
```

```
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

    public File copy() {
        File file = new File();
        file.data = this.data;
        file.name = this.name + " (copy)";
        return file;
    }
}
```

5. Перегрузка методов и конструкторов

Перегрузить метод (overload) – это значит определить несколько одинаковых методов с разным набором параметров.

Например:

```
class Math {  
    public int sum(int a, int b) {  
        return a + b;  
    }  
  
    public int sum(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

В примере показаны два метода, которые вычисляют сумму числовых аргументов. Тип возвращаемого значения и имена методов одинаковые, но набор аргументов отличается. При вызове метода `sum`, в зависимости от типов и количества переданных аргументов, будет вызвана та или иная реализация метода `sum`. Необходимо обратить внимание на то, что при перегрузке методов имена аргументов не имеют значения, роль играет только тип и количество параметров.

Ошибка при перегрузке:

```
public int sum(int a, int b) {  
    return a + b;  
}  
  
public int sum(int x, int y) {  
    return a + b;  
}
```

При перегрузке методов можно изменять тип, не меняю количество аргументов:

```
public int sum(int a, int b) {  
    return a + b;  
}  
  
public float sum(float x, float y) {  
    return a + b;  
}
```


При перегрузке конструкторов используются те же правила, что и при перегрузке методов. Если мы определили один конструктор с параметрами, конструктор по умолчанию работать не будет и даст ошибку, для того, чтобы при наличии других версий конструкторов, конструктор по умолчанию работал, его необходимо явно объявить (можно оставить тело пустым или определить значения по умолчанию):

```
public class Main {  
    public static void main(String[] args) {  
        Math math1 = new Math();  
        math1.x = 5;  
        math1.y = 10;  
        System.out.println("Max from: " + math1.x + " " + math1.y + " = " + math1.max());  
        Math math2 = new Math(2, 3);  
        System.out.println("Min from: " + math2.x + " " + math2.y + " = " + math2.min());  
        System.out.println("Sum: " + math2.x + " " + math2.y + " = " + math2.sum());  
    }  
}
```

```
class Math {  
    int x, y;  
  
    public Math() {  
        this.x = 0;  
        this.y = 0;  
    }  
    public Math(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int sum() {  
        return x + y;  
    }  
  
    public int min() {  
        return x <= y ? x : y;  
    }  
  
    public int max() {  
        return x >= y ? x : y;  
    }  
}
```

6. Наследование

Благодаря наследованию класс наследник получает доступ к полям и методам базового класса (класса родителя). Наследовать можно классы, а можно интерфейсы (рассмотрим позже). Для того, чтобы класс В наследовал от класса А, при объявлении класса В следует указать ключевое слово `extends` и имя базового класса (А).

Пример:

```
public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.sum();
        b.sum();
    }
}

class A {
    int x = 5, y = 5;
    public void sum() {
        System.out.println(x + y);
    }
}

class B extends A {
}
```

Как видно из примера, хотя класс В пустой, но благодаря наследованию, он получил доступ к полям `x`, `y` и методу `sum()`.

Обычно, классы наследники либо дополняют базовый класс, либо переопределяют методы базового класса. Если дополнение – это простое добавление новых полей и методов, то процесс переопределения необходимо рассмотреть подробнее. Переопределись метод (`override`) – это значит обеспечить специфическую реализацию метода, отличающуюся от реализации одноименного метода базового класса.

Рассмотри пример:

```
public class Main {
    public static void main(String[] args) {
```

```

    Square square = new Square(4);
    Rectangle rectangle = new Rectangle(3, 5);
    System.out.println("Square perimeter: " + square.getPerimeter());
    System.out.println("Square area: " + square.getArea());
    System.out.println("Rectangle perimeter: " + rectangle.getPerimeter());
    System.out.println("Rectangle area: " + rectangle.getArea());
}
}

```

```

class Square {
    int width;

    public Square(int width) {
        this.width = width;
    }
    public int getPerimeter() {
        return 4 * width;
    }
    public int getArea() {
        return width * width;
    }
}

```

```

class Rectangle extends Square {
    int height;

    public Rectangle(int width, int height) {
        super(width);
        this.height = height;
    }
    @Override
    public int getPerimeter() {
        return 2 * (height + width);
    }
    @Override
    public int getArea() {
        return width * height;
    }
}

```

Класс квадрат имеет поле width и методы для расчета периметра и площади квадрата. От этого класса наследуется класс прямоугольник. В нем добавлено поле height и предопределены методы расчета периметра и площади (т.к.

расчетные формулы квадрата и прямоугольника отличаются). Аннотация `@Override` указывает, что метод был предопределен. Обратите внимание на выделенную строку в конструкторе класса `Rectangle`. Ключевое слово `super` обеспечивает доступ к суперклассу, т.е. можно использовать реализацию методов базового класса внутри методов класса наследника. Если использовать запись `super(<аргументы>)`, то это обеспечивает доступ к конструктору суперкласса. Вызов конструктора суперкласса обязательно должен происходить в конструкторе класса наследника. Следующая форма ключевого слова `super` – это доступ к члену суперкласса, например: `super.x` или `super.x()`;

Для проверки, какому классу принадлежит объект (с учетом наследования) используйте оператор `instanceof`.

Пример:

```
if(object instanceof Square){
    System.out.println("Object type is Square");
}
else if(object instanceof Rectangle){
    System.out.println("Object type is Rectangle");
}
```

В языке Java существует возможность создавать базовые классы, которые не полностью реализованы, а требуют дополнительной реализации в классах наследниках. Такие классы называются **абстрактными**. Т.к. абстрактный класс не реализован до конца, то создавать экземпляры такого класса нельзя. Для определения абстрактного класса используйте ключевое слово `abstract`:

```
abstract class A {
}
```

Методы абстрактного класса могут иметь реализацию (как в обычных классах), но также могут быть абстрактными, такие методы должен обязательно реализовать класс наследник. Для определения абстрактного метода используйте аналогичное ключевое слово `abstract`:

```
public abstract void someMethod();
```

7. Понятие интерфейса

Интерфейс определяет как элементы будут взаимодействовать между собой. Все поля, которые объявлены в интерфейсе автоматически являются `static final` (статическими константами). Все члены класса, как и методы, неявно объявляются `public`. Интерфейс можно сравнить с полностью абстрактным классом. Для создания интерфейса используйте ключевое слово `interface`.

В языке Java присутствует множественное наследование интерфейсов, в отличие от классов (может быть только один класс родитель). Для наследования от интерфейса используйте ключевое слово `implements`. Если вы наследуете несколько интерфейсов, тогда укажите их через запятую. Любой класс одновременно может наследоваться от класса и от интерфейсов.

Рассмотрим пример работы с интерфейсом. Изменим код программы вычисления периметра и площади фигур:

```
public class Main {
    public static void main(String[] args) {
        Square square = new Square(4);
        Rectangle rectangle = new Rectangle(3, 5);
        System.out.println("Square perimeter: " + square.getPerimeter());
        System.out.println("Square area: " + square.getArea());
        System.out.println("Rectangle perimeter: " + rectangle.getPerimeter());
        System.out.println("Rectangle area: " + rectangle.getArea());
    }
}

interface Shape {
    int getPerimeter();
    int getArea();
}

class Square implements Shape {
    int width;
    public Square(int width) {
        this.width = width;
    }
    @Override
    public int getPerimeter() {
        return 4 * width;
    }
}
```

```

@Override
public int getArea() {
    return width * width;
}
}

class Rectangle implements Shape {
    int width;
    int height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    @Override
    public int getPerimeter() {
        return 2 * (height + width);
    }
    @Override
    public int getArea() {
        return width * height;
    }
}

```

Если класс Rectangle унаследовать от Square, он не будет обязательно нуждаться в реализации методов интерфейса Shape, т.к. они уже определены в классе Square. Если вы хотите чтобы класс обязательно определял собственную реализацию тех или иных методов, унаследуйте его от интерфейса или базового класса. Игнорирования определения абстрактных методов вызовет ошибку.

8. Вложенные и внутренние классы

В языке Java существуют вложенные (nested) и внутренние (inner) классы. Все они обязательно определяются внутри другого класса. Область видимости такого класса ограничена внешним классом. Внутренние классы следует применять, потому что они повышают структурирование и улучшение читаемости кода. Также, это хороший способ группировки классов. Если классы помощники используются только в связке с другими классами, нет смысла определять их в отдельных файлах.

В свою очередь, внутренние классы бывают статическими (вложенными) и нестатическими (внутренними). Статические вложенные классы применяются тогда, когда не нужна связь между объектами внутреннего и внешнего классов. Для создания экземпляра такого класса не нужен объект внешнего класса и нельзя обращаться к нестатическим полям и методам внешнего класса.

Создание объекта вложенного класса:

```
OuterClass.NestedClass object = new OuterClass.NestedClass();
```

Внутренний класс имеет доступ ко всем переменным и методам своего внешнего класса и может ссылаться на них. Внутренние классы не могут существовать без экземпляра внешнего класса.

Создание объекта внутреннего класса:

```
OuterClass.InnerClass object = outerObject.new InnerClass();
```

9. Анонимные классы

Анонимные классы – это безымянные классы, которые декларируются внутри методов основного класса и могут быть использованы только внутри этих методов. Главное требование к анонимному классу – он должен наследовать существующий класс или реализовывать существующий интерфейс.

Рассмотрим работу с анонимными классами на примере:

```
public class Main {
    public static void main(String[] args) {
        ArrayUtils uArray = new ArrayUtils(50);
        uArray.fill(3);
        uArray.print();
        uArray.sum(new ArrayUtils.OnCalculationListener() {
            @Override
            public void getResult(int sum) {
                System.out.println("Array sum = " + sum);
            }
        });
    }
}

class ArrayUtils {
    private int[] array;

    public ArrayUtils(int count) {
        array = new int[count];
        fill(1);
    }

    public void fill(int value) {
        Arrays.fill(array, value);
    }

    public void print() {
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");
        System.out.println();
    }

    public void sum(OnCalculationListener calculationListener) {
        int sum = 0;
```



```

    for (int i = 0; i < array.length; i++)
        sum += array[i];
    if (calculationListener != null)
        calculationListener.getResult(sum);
}

interface OnCalculationListener {
    void getResult(int sum);
}
}

```

Метод `sum(OnCalculationListener calculationListener)`, класса `ArrayUtils`, принимает в качестве параметра объект интерфейса (или так называемого слушателя), интерфейс описывает метод `getResult(int sum)`, т.е. метод `sum` предполагает, что клиент реализует метод интерфейса и определит в нем логику для работы с результатом суммирования элементов массива. В методе `main(String[] args)` мы вызвали метод `uArray.sum` и анонимно реализовали класс наследник интерфейса `OnCalculationListener`. В методе `getResult(int sum)` мы выводим результат суммирования в консоль. Возникает вопросы, мы ведь могли определить у метода `sum(OnCalculationListener calculationListener)` возвращаемое значение и вернуть сумму при вызове этого метода. Минус такого подхода в том, что он работает последовательно, пока идут вычисления, клиентская программа будет ожидать результат. Если элементов массива слишком много, тогда бы мы реализовали отдельный поток, в котором происходили вычисления, а слушатель асинхронно принимал результат. Также, возвращаемое значение позволяет вернуть только один результат, а так возможно определить в интерфейсе несколько методов, которые, в зависимости от входных данных, будут обрабатываться клиентом.

10. Шаблоны (Generics)

Благодаря шаблонам в языке Java есть возможность создавать классы, которые будут приспособлены под работу с разными типами. Эти классы называют шаблонными или параметризованными. Для создания такого класса, необходимо определить в угловых скобках формальный тип и указать его у полей класса. Также, формальный тип можно указывать как тип возвращаемого значения или тип аргументов в методах такого класса. Вместо формального параметра, если от него не зависят другие аргументы и возвращаемое значение, есть возможность указывать неизвестный параметр `<?>` или неизвестный параметр ограниченного типа `<? extends A>` (т.е. все классы наследники класса A).

Рассмотрим пример работы с шаблонными классами:

```
public class Main {
    public static void main(String[] args) {
        VirtualCollection<String> collection = new VirtualCollection<String>();
        collection.add("Привет");
        collection.add("Шаг");
        collection.add("и");
        collection.add("весь");
        collection.add("Мир!");
        collection.print();
        System.out.print("Второй элемент: " + collection.get(1));
    }
}
```

```
class VirtualCollection<T extends Object> {
    private T[] array;
    public VirtualCollection() {
        array = (T[]) new Object[0];
    }

    public void print() {
        for (T anArray : array) System.out.print(anArray + " ");
        System.out.println();
    }

    public void add(T newValue) {
        T[] newArray = Arrays.copyOf(array, array.length + 1);
        this.array = newArray;
        array[array.length - 1] = newValue;
    }
}
```

```
public T get(int position) {  
    if (position >= 0 && position < array.length)  
        return array[position];  
    else return null;  
}  
  
}
```

Мы создали примитивную коллекцию элементов, которая, в отличие от массива, умеет менять свою длину (метод add(T)).

При создании объекта шаблонного класса мы обязательно должны указать в угловых скобках конкретный тип данных. Мы выбрали строковый тип данных, занесли несколько элементов и отобразили результат на экран.