

# Java OOP

Курс Java OOP — раскрывает концепцию объектно ориентированного программирования. Рассматривается ее реализация в языке программирования Java. Также рассматривается инструментарий стандартной библиотеки JDK и способы организации работы с ним.

Киев 2016

**Составил: Цымбалюк А.Н.**

# Структура и описание курса

Курс предназначен для тех, кто **успешно усвоил** курс **Java Start** и желает продолжить изучение программирования на Java.

Курс структурирован по изучаемым темам. Это поможет с построением логической картины принципов ООП, и их использования. В конце каждого основного раздела дано «домашнее задание», выполнение которого будет способствовать закреплению пройденного материала.

Курс снабжен как теоретическим материалом, так и большим количеством примеров практической реализации той или иной задачи.

Также в курсе описано много дополнительного материала, не вошедшего в основной курс, но играющего важную роль в современной разработке. Однако при первом чтении его можно опустить без ущерба для понимания дальнейшего материала.

Надеемся, этот курс поможет вам в изучении Java и станет хорошим подспорьем для дальнейшего развития.

Для связи с автором данного курса можно использовать e-mail:  
[tsimbalukalexander@gmail.com](mailto:tsimbalukalexander@gmail.com)

# Оглавление. Часть I

1. Введение в ООП
2. Дополнительный материал (внутренние классы)
3. Дополнительный материал (Mutable и Immutable object)
4. Наследование
5. Дополнительный материал (анонимные классы)
6. Дополнительный материал (Перечисления)
7. Полиморфизм и работа с исключениями
8. Дополнительный материал (механизм рефлексии)
9. Дополнительный материал (аннотации)
10. Интерфейсы
11. Дополнительный материал (функциональные интерфейсы)
12. Потоки ввода вывода
13. Дополнительный материал (NIO 2)
14. Многопоточное программирование. Часть I
15. Дополнительный материал (Executors и интерфейсы Callable, Future)
16. Многопоточное программирование. Часть II
17. Дополнительный материал (Синхронизаторы)
18. Класс `java.lang.Object`

## Оглавление. Часть II

19. Дополнительный материал (Маршалинг, Демаршалинг в XML и JSON)
20. Generics и интерфейс Collection
21. Дополнительный материал (Stream API)
22. Map
23. Дополнительный материал (Map в JDK 1.8. и Optional)
24. Сетевое программирование на Java
25. Проект сетевого чата на сокетах.
26. Final

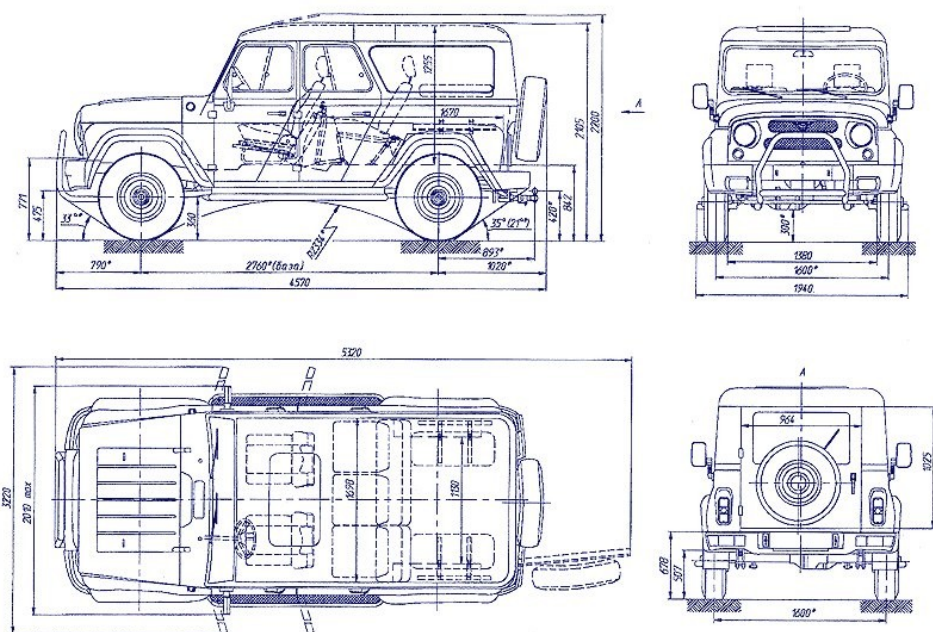
# Java OOP

## (Введение в ООП)

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия **объектов** и **классов**.

**Класс** является описываемой на языке терминологии исходного кода моделью еще не существующей сущности (объекта). Фактически он описывает устройство объекта, являясь своего рода чертежом.

# Класс



# Объект



Класс → Объект

Класс ≠ Объект





## Объект:

### Свойства:

- Цвет
- Масса
- Год выпуска
- Скорость

### Методы:

- Сигнал
- Ускорение
- Торможение

Свойства + Методы = Члены класса (объекта)

**class** – КЛЮЧЕВОЕ СЛОВО

## Объявление класса в Java

**class** Car{  
String color;  
**double** weight;  
**int** year;  
**private double** velocity=0;

Свойства

```
void beep(){  
    System.out.println("BEEP-BEEP!!!");  
}  
void acceleration (double a){  
    velocity=velocity+a;  
}  
void deceleration(double b){  
    if(velocity-b>=0) velocity=velocity-b;  
}  
void print(){  
    System.out.println("Color: "+color);  
    System.out.println("Weight: "+weight);  
    System.out.println("Year car: "+year);  
    System.out.println("Velocity: "+velocity);  
}  
}
```

Методы



# Создание объекта на основе описанного класса

```
package com.gmail.tsa;
```

```
class Car{  
    String color;  
    double weight;  
    int year;  
    private double velocity=0;  
    void beep(){  
        System.out.println("BEEP-BEEP!!!");  
    }  
    void acceleration (double a){  
        velocity=velocity+a;  
    }  
    void deceleration(double b){  
        if(velocity-b>=0) velocity=velocity-b;  
    }  
    void print(){  
        System.out.println("Color: "+color);  
        System.out.println("Weight: "+weight);  
        System.out.println("Year car: "+year);  
        System.out.println("Velocity: "+velocity);  
    }  
}
```

Описание класса




```
public class Main {
```

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

```
        Car m1=new Car();  
        m1.color="gren";  
        m1.weight=1500;  
        m1.year=1982;  
        m1.print();  
        m1.acceleration(30);  
        m1.print();
```

Создание объекта  
Установка его свойств  
Использование его методов



```
    }
```

Имя создаваемого объекта

**new** - Ключевое слово

Параметры конструктора  
(если они есть)

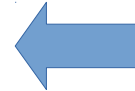
Car car=**new** Car();

На основе какого класса создается объект

**Для обращения к свойствам или методам объектов используется оператор «.»**

```
car.color="green";  
car.weight=1500;  
car.year=1982;
```

Установка свойств  
объекта



```
car.print();  
car.acceleration(30);  
car.print();
```

Вызов методов  
объекта



# Модификаторы доступа:

- **public** — члены класса доступны всем
- **protected** — члены класса доступны внутри пакета и в наследниках
- **default** — члены класса видны внутри пакета
- **private** — члены класса доступны только внутри класса.

# Некоторые ключевые слова:

- **static:**
  - данные, которые должны существовать в единственном экземпляре на всю программу, а не в каждом объекте.
  - методы, которые не оперируют ни с какими данными (полями) объекта.
- **this:** указывает на объект, из которого вызывается.
- **finalize():** — метод, который выполнится перед удалением объекта.

# Конструкторы

Имя конструктора должно совпадать с именем класса!!!



```
Car(String color, double weight, int year) {  
    this.color=color;  
    this.weight=weight;  
    this.year=year;  
}
```

Определение конструктора должно идти после свойств класса.



# Порядок описания элементов в классе

Сначала описываются статические свойства класса

1. static:

- public
- protected
- default
- private

2. Не статические свойства класса:

- public
- protected
- default
- private

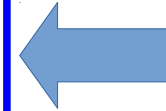
Конструкторы класса

Методы класса

## Описание класса с конструктором

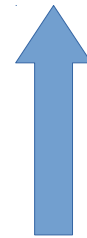
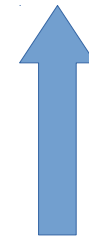
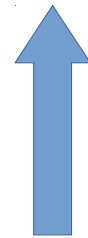
```
class Car{
    String color;
    double weight;
    int year;
    private double velocity=0;
    Car(String color, double weight,int year){
        this.color=color;
        this.weight=weight;
        this.year=year;
    }
    void beep(){
        System.out.println("BEEP-BEEP!!!");
    }
    void acceleration (double a){
        velocity=velocity+a;
    }
    void deceleration(double b){
        if(velocity-b>=0) velocity=velocity-b;
    }
    void print(){
        System.out.println("Color: "+color);
        System.out.println("Weight: "+weight);
        System.out.println("Year car: "+year);
        System.out.println("Velocity: "+velocity);
    }
}
```

Конструктор



Создание объекта  
при использовании конструктора с параметрами

```
Car car=new Car( "Green", 1500, 1986 );
```



Параметры, передаваемые конструктору

Работа с объектами схожа с работой с ссылочными типами:

**Объект может быть :**

- **Передан** в качестве аргумента методу;
- **Возвращен** в результате работы метода;
- Элементом массива;
- Составной частью другого объекта;

Объект как результат работы метода



Объект как аргумент метода



```
Car repaintCar(Car a, String newcolor){  
    a.color=newcolor;  
    return a;  
}
```

Класс, метод которого возвращает  
переменную этого класса

```
class Car{
    String color;
    double weight;
    int year;
    private double velocity=0;
    Car(String color, double weight,int year){
        this.color=color;
        this.weight=weight;
        this.year=year;
    }
    void beep(){
        System.out.println("BEEP-BEEP!!!");
    }
    void acceleration (double a){
        velocity=velocity+a;
    }
    void deceleration(double b){
        if(velocity-b>=0) velocity=velocity-b;
    }
    void print(){
        System.out.println("Color: "+color);
        System.out.println("Weight: "+weight);
        System.out.println("Year car: "+year);
        System.out.println("Velocity: "+velocity);
    }
}
```

```
Car repaintCar(Car a, String newcolor){
    a.color=newcolor;
    return a;
}
```

Метод работающий  
с объектами



```
}
```

# Пример использования метода работающего с объектами

```
public class Main {  
  
    public static void main(String[] args) {  
        Car car=new Car("Green",1500,1986);  
        car.acceleration(30);  
        car=car.repaintcar(car, "Red");  
        car.print();  
    }  
}
```

## Вывод программы

```
Color: Red  
Weight: 1500.0  
Year car: 1986  
Velocity: 30.0
```



## Немного об инкапсуляции данных

Данные, которые нужно «скрыть» от пользователя, следует объявить с ключевым словом `private` или `protected`.

Для того, чтобы получить возможность считывать или чтобы процесс установки значений скрытых данных был контролируемым, создают публичные функции по возврату и установке значений.

**!** Желательно (но не обязательно) чтобы функции, которые возвращают значение, имели префикс "**get**" в названии, а те, что устанавливают - значение со словом "**set**".

## Типичное объявление метода get

```
Тип_переменной_которую_необходимо_вернуть get  
имя_переменной(){  
    return переменная_которую_необходимо_вернуть  
}
```

Предположим, что необходимо получить значение скрытой переменной с именем name типа String

```
public String getName(){  
    return name;  
}
```

## Типичное объявление метода set

```
void set имя_переменной(Тип_переменной_которую_необходимо  
установить имя_переменной ){  
    this.имя_переменной = имя_переменной;  
}
```

Предположим, что нужно установить значение скрытой переменной с именем name типа String

```
public void setName(String name){  
    this.name = name;  
}
```

## Краткие итоги урока

**Класс** — пользовательский тип данных. Он состоит из свойств и методов класса. Объекты (по сути обычные переменные, но уже пользовательского типа данных) относятся к ссылочному типу данных.

Для более удобного способа инициализации свойств класса используют **конструкторы**. Конструкторы — это методы, которые не имеют типа возвращаемого значения, и чье название обязательно совпадает с именем класса.

В классе существует установленный порядок объявления свойств, конструкторов и методов класса.

Для реализации принципов инкапсуляции данных используют модификаторы доступа **private, protected, public**. Желательно чтобы все свойства класса были закрытыми (**private** или **protected**) и доступ к ним осуществлялся с помощью специальных **public**-методов (сеттеры и геттеры).

## Дополнительная литература по теме данного урока.

- Герберт Шилд - Java. Полное руководство. 8-е издание. стр. 145-162.
- Кей Хорстман, Гари Корнелл — Библиотека профессионала Java Том 1. Основы. стр. 149-179.

## Домашнее задание:

- 1) Описать класс «Cat» (в качестве образца можно взять домашнего питомца). Наделить его свойствами и методами. Создать несколько экземпляров объектов этого класса. Использовать эти объекты.
- 2) Описать класс «Triangle». В качестве свойств возьмите длины сторон треугольника. Реализуйте метод, который будет возвращать площадь этого треугольника. Создайте несколько объектов этого класса и протестируйте их.
- 3) Описать класс «Vector3d» (т. е., он должен описывать вектор в трехмерной, декартовой системе координат). В качестве свойств этого класса возьмите координаты вектора. Для этого класса реализовать методы сложения, скалярного и векторного произведения векторов. Создайте несколько объектов этого класса и протестируйте их.
- 4) Опишите класс Phone (одним из свойств должен быть его номер). Также опишите класс Network (сеть мобильного оператора). Телефон должен иметь метод регистрации в сети мобильного оператора. Также у телефона должен быть метод call (номер другого телефона), который переберет все телефоны, зарегистрированные в сети. Если такой номер будет найден, то осуществить вызов, если нет - вывести сообщение об ошибочности набранного номера.



# Внутренние классы

## Дополнительный материал к лекции «Знакомство с классами»

Определение класса может быть выполнено в пределах определения другого класса.

Такие классы называются внутренними (inner class).

## Виды внутренних классов и область их видимости

Внутренние классы подразделяются на два подтипа:

- Статические внутренние классы;
- Не статические внутренние классы;

Не статические внутренние классы создаются на основе существующего объекта внешнего класса. Внутренний класс имеет полный доступ к свойствам внешнего класса (в том числе и `private`-переменным).

Для создания статических внутренних классов достаточно использовать только название внешнего класса (т. е., объект внешнего класса создавать не обязательно). Статический внутренний класс имеет доступ только к статическим свойствам внешнего.

## Пример использования не статического внутреннего класса

```
package com.gmail.tsa;
```

```
public class IntList {  
    private int[] arr = new int[10];  
    private int n;
```

«Список» для хранения элементов  
типа int на основе массива

```
    public void add(int number) {  
        if (n == arr.length - 1) {  
            this.resize();  
        }  
        arr[n++] = number;  
    }
```

Метод по добавлению элемента,  
если элементов добавлено много,  
то увеличиваем размер массива

```
    public class IntIterator {  
        private int pointer = 0;  
        public boolean hasNext() {  
            return pointer < IntList.this.n;  
        }  
        public int next() {  
            return IntList.this.arr[pointer++];  
        }  
    }
```

Внутренний класс,  
предназначенный для прохода по  
структуре

```
    public IntIterator getIterator() {  
        return new IntIterator();  
    }
```

Метод для создания  
переменной внутреннего класса

```
    private void resize() {  
        int[] temp = new int[(this.arr.length * 7) / 4];  
        System.arraycopy(arr, 0, temp, 0, arr.length);  
        arr = temp;  
    }
```

Увеличиваем массив  
по мере  
необходимости

```
}
```

## Пример использования не статического внутреннего класса

```
package com.gmail.tsa;

import com.gmail.tsa.IntList.IntIterator;

public class Main {

    public static void main(String[] args) {

        IntList intList = new IntList();
        intList.add(3);
        intList.add(5);
        intList.add(8);
        intList.add(11);
        intList.add(3);
        intList.add(18);

        IntIterator iter = intList.getIterator();

        while (iter.hasNext()) {
            System.out.print(iter.next()+" ");
        }

    }
}
```

Создание переменной внешнего класса. Так как его свойства закрыты, то получить к ним доступ невозможно.

Добавление элементов

Создание переменной внутреннего класса, и использование его для доступа к данным внешнего.

Таким образом, одно из возможных применений внутренних классов - получение доступа к данным, которые необходимо получить с помощью объекта другого класса и никак более.

```
package com.gmail.tsa;
```

## Пример использования статического внутреннего класса

```
public class PsevdoMash {
```

```
    private static int getCube(int number) {  
        return number * number * number;  
    }
```

```
    private static int getSquare(int number) {  
        return number * number;  
    }  
}
```

private методы для инициализации  
ВНОВЬ создаваемых объектов

```
    public static class Cube {  
        private int number;  
        public Cube(int number) {  
            super();  
            this.number = PsevdoMash.getCube(number);  
        }  
        public int getNumber() {  
            return number;  
        }  
        public void setNumber(int number) {  
            this.number = number;  
        }  
    }
```

```
    public static class Square {  
        private int number;  
        public Square(int number) {  
            super();  
            this.number = PsevdoMash.getSquare(number);  
        }  
        public int getNumber() {  
            return number;  
        }  
        public void setNumber(int number) {  
            this.number = number;  
        }  
    }
```

```
}
```

Описание статических  
внутренних классов. Для  
их генерации используется  
внешний класс.

## Пример использования статического внутреннего класса

```
package com.gmail.tsa;

import com.gmail.tsa.PsevdoMash.Cube;
import com.gmail.tsa.PsevdoMash.Square;

public class Main {

    public static void main(String[] args) {

        Cube cube = new PsevdoMash.Cube(4);
        Square square = new PsevdoMash.Square(4);

        System.out.println(cube.getNumber());
        System.out.println(square.getNumber());
    }
}
```

Создание переменных  
внутренних классов, с  
использованием статических  
методов внешнего

Таким образом, одна из сфер использования статических внутренних классов — это генерация переменных разных классов на основе одного класса.



## Краткие итоги урока

**Внутренние классы** — инструмент, направленный на реализацию случая, когда одна сущность (описываемая внешним классом) должна управляться порождаемой ею сущностью (внутренним классом).

Внутренние классы делятся на два типа:

- **Не статические** — часто используют для управления или получения услуг.
- **Статические** — используют для того, чтобы подчеркнуть, что текущий класс является частью (более мелкой) внешнего.

## Дополнительная литература по теме данного урока.

- Брюс Эккель Философия Java. 4-е издание. стр. 245-276;
- Герберт Шилд Java. Полное руководство. 8-е издание. стр. 184-186;

# Mutable and immutable object

## Дополнительный материал к лекции

### «Введение в ООП »

Mutable (изменяемый) объект - это объект, свойства которого могут изменяться после его создания.

Immutable (неизменяемый) объект - это такой объект, чье внешнее видимое состояние не может измениться после его создания.

Составил: Цымбалюк А.Н.

## Mutable object

Mutable (далее по тексту — изменяемый) объект — это объект, свойства которого могут быть изменены после его создания. Т.е. это объект класса, в котором описаны свойства и методы (сеттеры и геттеры) для изменения этих свойств. При изменении свойств такого объекта в памяти не создается новый объект, а лишь изменяется значение его свойств.

К преимуществам объекта такого типа можно отнести простоту создания и высокую скорость выполнения изменения свойств.

Для создания изменяемого объекта:

- Опишите свойства класса с модификатором `private` или `protected`;
- Для каждого из них создайте методы получения и установки;

Значительная доля классов в Java описывает именно изменяемые объекты. Т.е., в контексте упоминания в Java, часто опускают слова изменяемый и просто называют объектом класса.

## Пример класса, который описывает изменяемый объект

```
package com.gmail.tsa;
```

```
public class MutablePoint {
```

```
    private double x;  
    private double y;
```

Свойства класса

```
    public MutablePoint(double x, double y) {  
        super();  
        this.x = x;  
        this.y = y;  
    }
```

```
    public double getX() {  
        return x;  
    }
```

```
    public void setX(double x) {  
        this.x = x;  
    }
```

```
    public double getY() {  
        return y;  
    }
```

```
    public void setY(double y) {  
        this.y = y;  
    }
```

Методы получения и установки свойств

```
@Override
```

```
    public String toString() {  
        return "MutablePoint [x=" + x + ", y=" + y + "];"  
    }
```

```
}
```

## Пример использования изменяемого объекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        MutablePoint point = new MutablePoint(2, 4);
```

```
        System.out.println(point);
```

```
        point.setX(5);
```

```
        System.out.println(point);
```

```
    }
```

```
}
```

Создание объекта изменяемого типа



Изменение свойств объекта



Т.е., объект изменяемого типа - это то, что привыкли называть просто объектом обычного класса.

# Immutable object

**Immutable** (в дальнейшем - неизменяемый) объект – это объект, состояние которого не может быть изменено после создания. Здесь состоянием объекта, по существу, считаются значения, хранимые в экземпляре класса, будь то примитивные типы или ссылочные типы.

Преимущества неизменяемых типов:

- Легко конструировать, тестировать и использовать;
- Автоматически потокобезопасны и не имеют проблем синхронизации;
- Не требуют конструктора копирования;
- Позволяют выполнить «ленивую инициализацию» хэшкода и кэшировать возвращаемое значение;
- Не требуют защищенного копирования, когда используются как поле;
- Делают хорошие Map ключи и Set элементы (эти объекты не должны менять состояние, когда находятся в коллекции);
- Делают свой класс постоянным, единожды создав его, он уже не нуждается в повторной проверке;
- Всегда имеют «атомарность по отношению к сбою». Если неизменяемый объект бросает исключение, он никогда не останется в нежелательном или неопределенном состоянии;

К недостаткам можно отнести медленную работу при изменении свойств, повышенное потребление памяти.

## Описание неизменяемого типа

Для описания класса при создании объектов неизменяемого типа:

- 1) Все его поля являются `final`;
- 2) Класс объявляется как `final`;
- 3) Ссылка `this` не должна пропасть во время конструирования;
- 4) Любые поля, содержащие ссылки на изменяемые объекты, например массивы, совокупности или изменяемые классы:
  - Являются приватными;
  - Никогда не возвращаются и никаким другим образом не становятся доступными вызывающим операторам;
  - Являются единственными ссылками на те объекты, на которые они ссылаются;
  - Не изменяют после конструирования состояние объектов, на которые они ссылаются;



## Пояснение требований к ссылочным свойствам неизменяемого типа

Группа требований для свойств ссылочного типа кажется сложной, но это, главным образом, означает, что если вы собираетесь хранить ссылку на массив или другие изменяемые объекты, вы должны обеспечить своему классу монопольный доступ к этому изменяемому объекту (так как в ином случае кто-то другой сможет изменить его состояние) и убедиться, что вы не модифицировали его состояние после конструирования. Это усложнение необходимо, чтобы позволить неизменяемым объектам хранить ссылки на массивы. Помните, что если ссылки на массив или другие изменяемые поля инициализируются аргументами, переданными конструктору, вам необходимо, на всякий случай скопировать, аргументы вызывающей стороны. Иначе, вы не можете быть уверены, что у вас монопольный доступ к массиву.

## Пример класса неизменяемого объекта

```
package com.gmail.tsa;
```

```
public final class ImmutablePoint {
```

```
    private final double x;  
    private final double y;
```

Описание свойств класса. Объявлено с модификатором final.

```
    public ImmutablePoint(double x, double y) {  
        super();  
        this.x = x;  
        this.y = y;  
    }
```

```
    public double getX() {  
        return x;  
    }
```

```
    public double getY() {  
        return y;  
    }
```

Методы доступа к свойствам. Обратите внимание, что нет сеттеров.

```
    @Override  
    public String toString() {  
        return "ImmutablePoint [x=" + x + ", y=" + y + "];"  
    }
```


```
}
```

## Прием для организации изменения свойств неизменяемого типа

Если, все же, нужно обеспечить возможность изменения свойств неизменяемого объекта, то одним из приемов, который может реализовать такую возможность может быть такой — при изменении свойства создается новый объект данного класса, свойства которого равны новому значению.

Пример сеттеров, которые реализуют указанный выше прием

```
public ImmutablePoint setX(double x) {  
    return new ImmutablePoint(x, this.y);  
}  
  
public ImmutablePoint setY(double y) {  
    return new ImmutablePoint(this.x, y);  
}
```



Теперь методы установки возвращают новые объекты этого класса

## Пример класса неизменяемого объекта с ВОЗМОЖНОСТЬЮ ИЗМЕНЕНИЯ СВОЙСТВ

```
package com.gmail.tsa;
```

```
public final class ImmuatablePoint {
```

```
    private final double x;  
    private final double y;
```

Описание свойств класса. Объявлено с модификатором final.

```
    public ImmuatablePoint(double x, double y) {  
        super();  
        this.x = x;  
        this.y = y;  
    }
```

```
    public double getX() {  
        return x;  
    }  
    public double getY() {  
        return y;  
    }
```

Методы получения значений свойств.

```
    public ImmuatablePoint setX(double x) {  
        return new ImmuatablePoint(x, this.y);  
    }  
    public ImmuatablePoint setY(double y) {  
        return new ImmuatablePoint(this.x, y);  
    }
```

Методы установки, которые возвращают новые объекты.

```
    @Override  
    public String toString() {  
        return "ImmuatablePoint [x=" + x + ", y=" + y + "];"  
    }
```

```
}
```

## Основные сведения о перечислениях

```
package com.gmail.tsa;
```

```
public class Main {
```

Создание объектов неизменяемого типа

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

```
    ImmutablePoint pointOne = new ImmutablePoint(2, 4);
```

```
    System.out.println(pointOne);
```

```
    ImmutablePoint pointTwo = pointOne.setX(4);
```

```
    System.out.println(pointTwo);
```

```
    System.out.println(pointOne == pointTwo);
```

```
}
```

```
}
```

Но, это уже разные объекты, что и покажет результат сравнения

Изменение свойств объекта, где результатом будет новый объект

## Итоги урока

**Mutable** объект — это объект, свойства которого могут быть изменены после его создания. Т.е., это объект класса, в котором описаны свойства и методы (сеттеры и геттеры) для изменения этих свойств. При изменении свойств такого объекта в памяти не создается нового объекта, а лишь изменяется значение свойств.

**Immutable** объект – это объект, состояние которого не может быть изменено после создания. Здесь состоянием объекта, по существу, считаются значения, хранимые в экземпляре класса, будь то примитивные типы или ссылочные типы.

**Immutable** - тип часто применяется для облегчения контроля в многопоточной среде, и для упрощения работы с системами, использующими кеширование и единственность образца.

К недостаткам неизменяемых типов можно отнести низкую скорость работы и повышенный расход памяти.

## Дополнительная литература по теме данного урока.

- 1) Блинов И.Н., Романчик В.С. Java. Методы программирования / И.Н. Блинов, В.С. Романчик. – Минск: Четыре Четверти, 2013. – с. 80.

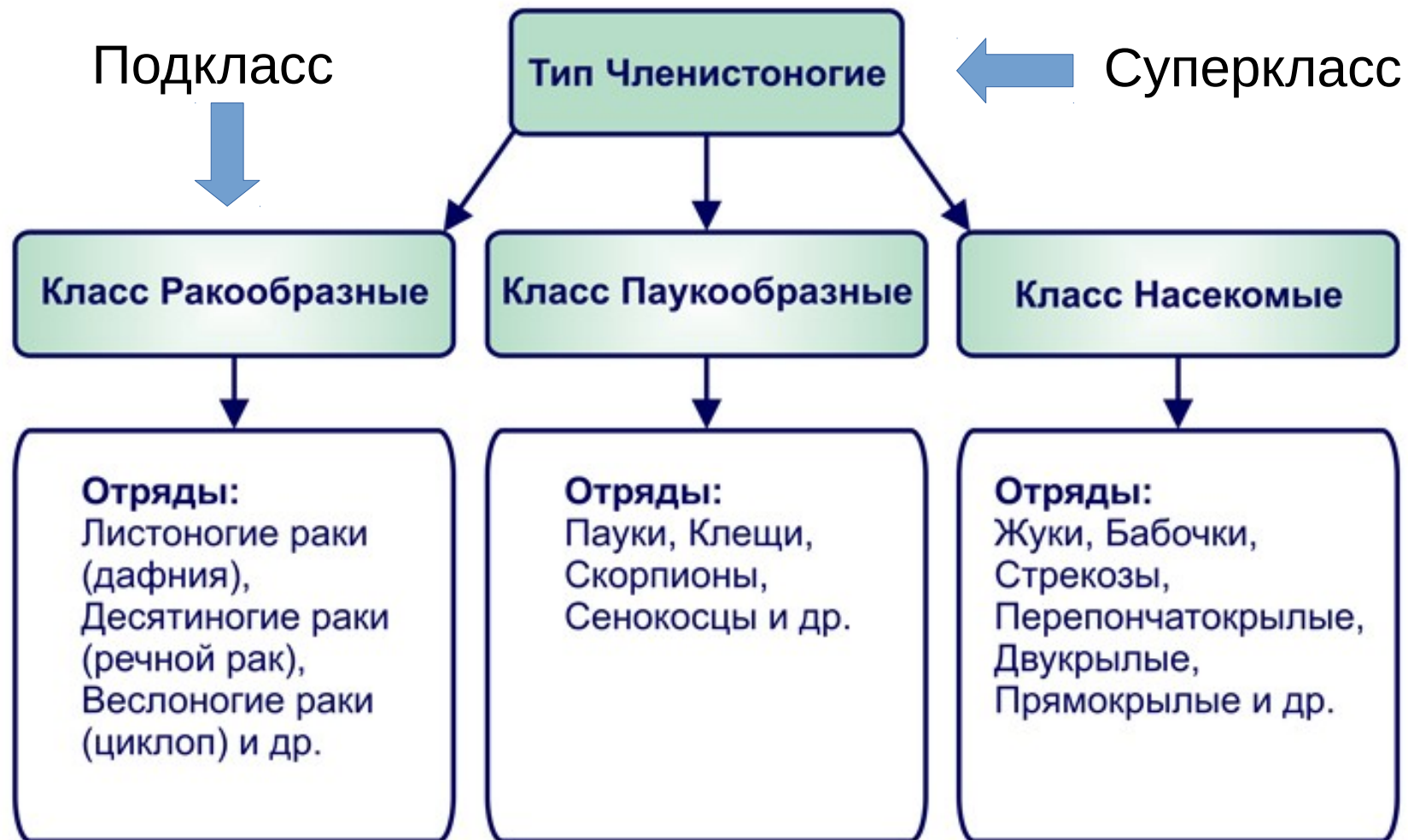
# Java OOP

## (Наследование)

**Наследование** — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов.



# Пример наследования в реальном мире



# Наследование в языке Java

Опишем класс Animals - он будет основой для классов, описывающих домашних животных

```
package com.gmail.tsa;

public class Animals {
    int age;
    double weight;
    boolean sex;
    String ration;

    public Animals(int age, double weight, boolean sex, String ration) {
        super();
        this.age = age;
        this.weight = weight;
        this.sex = sex;
        this.ration = ration;
    }

    public Animals() {
        super();
    }

    public void getVoice() {

    }

    public String toString() {
        return "age=" + age + ", weight=" + weight + ", sex=" + sex + ", ration=" +
            ration + "]";
    }
}
```

## Теперь на его основе можно создать класс Cat

Добавим коту кличку, породу и изменим метод голос.

Для проверки корректности использования наследования можно использовать следующее правило: если подкласс является суперклассом, то можно использовать механизм наследования.

Пример: Кот является животным, поэтому использование наследования обоснованно.

Для этого используем механизм наследования в Java

```
package com.gmail.tsa;
```

## Класс Cat - наследник Animals

**extends** - ключевое слово

```
public class Cat extends Animals {
```

```
    String name;  
    String type;
```

Добавление новых свойств

```
    public Cat(int age, double weigt, boolean sex, String ration, String  
        name, String type) {  
        super(age, weigt, sex, ration);  
        this.name = name;  
        this.type = type;
```

Вызов конструктора суперкласса

```
    }  
    @Override  
    public void getVoice() {  
        System.out.println("May - May");  
    }
```

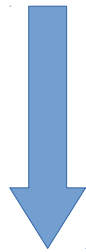
Изменение существующих методов

```
    @Override  
    public String toString() {  
        return "Cat [name=" + name + ", type=" + type + super.toString();  
    }
```

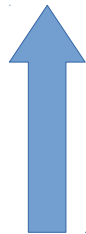
```
}
```

# Правило объявления подкласса

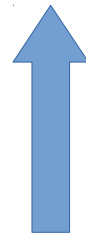
Ключевое слово, обозначающее наследование



```
class Cat extends Animals {...
```



Имя подкласса



Имя суперкласса

## **Подкласс может иметь следующие виды методов:**

1. Подменённые: новый класс не просто наследует метод суперкласса, но и дает ему новое определение.
2. Новые: новый класс приобретает совершенно новые методы.
3. Рекурсивные: новый класс просто наследует методы и переменные родительского класса.

## Модификаторы доступа свойств и методов класса

Modifier	Class	Package	Subclass	World
public	y	y	y	y
protected	y	y	y	n
no modifier	y	y	n	n
private	y	n	n	n

# О наследовании подробнее

Переменная суперкласса может ссылаться на объект подкласса

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Animals anim = new Animals(3, 4, true, "milk");
```

```
        Cat acat = new Cat(3, 4, true, "milk", "Barsic", "Angor");
```

```
        anim = acat; ◀ Переменная супер класса ссылается на объект подкласса
```

```
        voice(anim); ◀ Параметром метода является переменная суперкласса.  
        voice(acat); ◀ Однако, переменная подкласса автоматически приведется к ней.
```

```
    }
```

```
    public static void voice(Animals a) {
```

```
        a.getVoice();
```

```
    }
```

```
}
```

При таком присвоении суперкласс имеет доступ к членам определенным только в нем.



# Использование ключевого слова **super**

Для того, чтобы из подкласса обратиться к методу или конструктору суперкласса, используется ключевое слово **super**

Пример использования ключевого слова **super**

```
@Override  
public String toString() {  
    return "Cat [name=" + name + ", type=" + type + super.toString();  
}
```



Вызываем метод `toString()` суперкласса

# Использование `super` в конструкторе

С помощью `super` часто вызывают конструктор суперкласса. В таком случае параметры, которые нужно передать в конструктор суперкласса, записываются как параметры метода `super`.

Этот метод должен вызываться в конструкторе подкласса.

Пример использования **`super`** в конструкторе подкласса

```
public Cat(int age, double weight, boolean sex, String ration,  
String name, String type) {
```


```
    super(age, weight, sex, ration);
```

```
    this.name = name;
```

```
    this.type = type;
```

```
}
```

Использование  
конструктора суперкласса



# Переопределение методов

Если в классе наследнике - полное описание метода совпадает с методом суперкласса, то говорят о **переопределении** метода.

Если в вашем классе есть переопределенные методы, то их желательно (но не обязательно) пометить аннотацией `@Override`.

`@Override`



Аннотация

```
public void getVoice() {  
    System.out.println("May - May");  
}
```



Переопределенный метод

Аннотация `@Override` – средство контроля кода. Метод, для которого она объявлена, обязательно должен переопределять метод суперкласса.

# Использование абстрактных классов

**Абстрактный класс** — базовый класс, который не предполагает создания экземпляров. Абстрактный класс может содержать (и не содержать) абстрактные методы и переменные.

! Абстрактный метод содержит только описание метода, но не содержит тела. Если в классе хоть один метод является абстрактным, то и весь класс является абстрактным

Объявление абстрактного класса и метода должно начинаться ключевым словом **abstract**.

# Пример использования абстрактного класса

Сначала объявим абстрактный класс, который представляет двумерные фигуры. Единственные методы одинаковые для всех фигур - это вычисления периметра и площади.

```
package com.gmail.tsa;
```

```
public abstract class Shape {
```

Объявление абстрактного класса

```
    abstract double calculatePerimetr();
```

```
    abstract double calculateArea();
```

```
}
```

Объявление абстрактных методов

## Теперь создадим класс треугольник

```
package com.gmail.tsa;
```

```
public class Triangular extends Shape{
```

```
    double a;
```

```
    double b;
```

```
    double c;
```



Наследование абстрактного класса

```
@Override
```

```
double calculatePerimetr(){
```

```
    return a+b+c;
```

```
}
```



Реализация абстрактных методов

```
@Override
```

```
double calculateArea(){
```


```
    double polP=(a+b+c)/2;
```

```
    return Math.sqrt(polP*(polP-a)*(polP-b)*(polP-c));
```

```
}
```

```
}
```

Переменная абстрактного класса может ссылаться на объект класса, который его наследует.

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        Triangular tr=new Triangular();  
        tr.a=3;  
        tr.b=4;  
        tr.c=5;  
  
        Shape sp=tr;   
  
        System.out.println(sp.calculateArea());  
    }  
}
```

Присвоение переменной абстрактного класса объекту, реализующего его, класса.

Внимание! Такая переменная имеет доступ только к реализованным абстрактным методам.

## Использование ключевого слова **final**

Использование ключевого слова **final** перед определением метода запрещает его переопределение.

Использование ключевого слова **final** перед определением класса запрещает его наследование.



# Класс Object

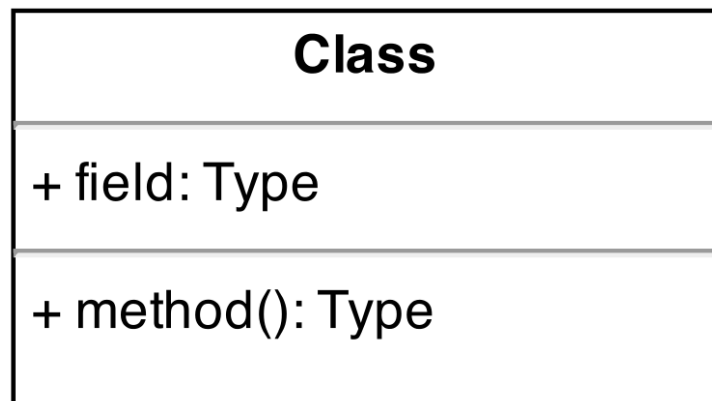
Все классы в Java являются подклассами суперкласса Object

Это значит, что ссылочная переменная класса может ссылаться на любой объект любого класса.

Методы, определенные в классе Object	Описание метода
Object clone()	Создает новый объект копированием
boolean equals(Object object)	Сравнивает объекты
void finalize()	Вызывается перед удалением объекта
class <?> getClass()	Получает класс объекта
int hashCode()	Возвращает хеш код объекта
void notify()	Возобновляет выполнение потока
void notifyAll()	Возобновляет выполнение всех потоков
String toString()	Возвращает строку, которая описывает объект
void wait	Ожидание другого потока выполнения
void wait(long )	Ожидание другого потока выполнения

# Основы UML диаграмм

Каждый класс может быть представлен в графическом виде



Имя класса

Описание свойств

Описание методов

+ public  
# protected  
- private

Кванторы видимости

## Для свойств

<квантор видимости><имя свойства> <тип>

## Для методов

<квантор видимости><имя метода>(список параметров)<тип>:

# Основы UML диаграмм

## Отношения между классами

————▷ Наследование

◊———— Агрегация

◆———— Композиция

-----▷ Реализация

————➔ Ассоциация

-----➔ Зависимость

! Удобный сайт для онлайн создания UML диаграмм  
[www.draw.io](http://www.draw.io)

# Наследование

## Отношения между классами

**Наследование** - показывает, что один из двух связанных классов (подкласс) является частной формой другого (суперкласса), который называется обобщением первого. На практике это означает, что любой экземпляр подкласса является также экземпляром суперкласса.

## Пример

Класс Cat является подклассом Animal. Так как кот является животным, то это отношение наследования.

## Графическое представление



Наследование

# Агрегация

## Отношения между классами

**Агрегация** — отношение «часть-целое» между двумя равноправными объектами, когда один объект (контейнер) имеет ссылку на другой объект. Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое — нет.

## Пример

Класс Автомобиль агрегирует классы Болт. Так как Автомобиль состоит из Болтов. В то же время Болт может существовать и вне автомобиля.

## Графическое представление

 Агрегация

# Композиция

## Отношения между классами

**Композиция** — более строгий вариант агрегирования, когда включаемый объект может существовать только как часть контейнера. Если контейнер будет уничтожен, то и включенный объект тоже будет уничтожен.

## Пример

Университет состоит из факультетов, однако факультет не существует вне университета.

## Графическое представление

◆ — Композиция

# Реализация

## Отношения между классами

**Реализация** — отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). Реализация — отношение целое-часть. Поставщик, как правило, является абстрактным классом или классом-интерфейсом.

## Пример

В классе Треугольник (смотри слайд посвященный абстрактным классам) реализованы методы вычисления площади и периметра абстрактного класса Shape

## Графическое представление

-----▷ Реализация

# Ассоциация

## Отношения между классами

**Ассоциация** - показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому. Является общим случаем композиции и агрегации.

## Пример

Класс Человек и класс Школа имеют ассоциацию, так как человек может учиться в школе.

## Графическое представление

————→ Ассоциация



# Зависимость

## Отношения между классами

**Зависимость** — это слабая форма отношения использования, при котором изменение в спецификации одного влечет за собой изменение другого, причем обратное не обязательно. Возникает, когда объект выступает, например, в форме параметра или локальной переменной.

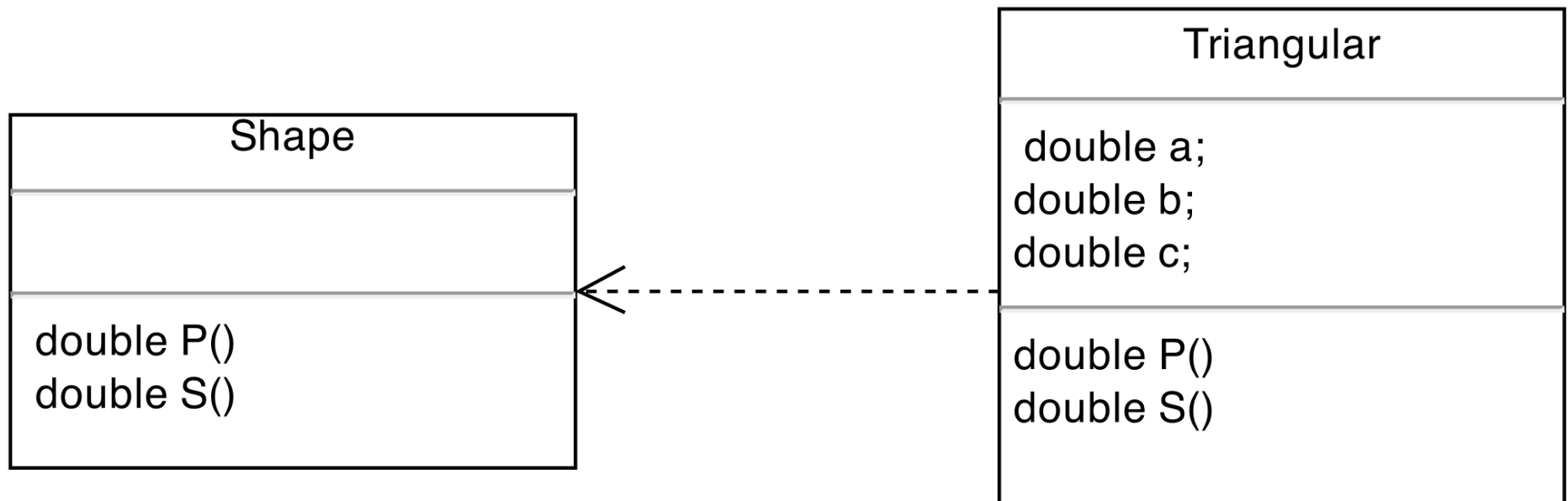
## Пример

Краткое содержание книги зависит от текста книги

## Графическое представление

-----> Зависимость

# Пример UML диаграмм двух классов



## Краткие итоги урока

**Наследование** — механизм, дающий возможность создавать новые классы на основе существующих.

Класс, который был базой для создания нового — **суперкласс**. Производный от базового — **подкласс**. Ключевое слово для описания того, что используется механизм наследования - **extends**. Все классы в Java неявно наследуют **Object**.

Таким образом подкласс получает от суперкласса все открытые свойства и методы (Внимание! К private свойствам и методам он доступа не имеет).

При создании объекта подкласса в памяти сначала создается объект суперкласса. Поэтому нужно выстраивать «лестницу конструкторов».

Подкласс всегда можно привести к суперклассу. Если методы подкласса переопределяют методы супер класса, то вариант вызываемого метода определится типом объекта.

**Абстрактные** классы (класс, в котором есть методы без реализации) - удобный способ построения архитектуры приложения. Для реализации абстрактного класса нужно переопределить все абстрактные классы. Реализующий класс также приводится к абстрактному.

## Дополнительная литература по теме данного урока.

- Брюс Эккель Философия Java. 4-е издание. стр. 169 - 197
- Герберт Шилд Java. Полное руководство. 8-е издание. стр. 195 - 218

## Домашнее задание

1. Создайте абстрактный класс Shape, в котором есть два абстрактных метода `double getPerimetr()` и `double getArea()`.
2. Создайте класс Point, в котором есть два свойства `double x` и `double y`.
3. Создайте классы, которые описывают, как минимум, три геометрические фигуры (они должны быть подклассами Shape). При этом они в качестве свойств должны содержать классы Point.
4. Создайте класс доска. Доска поделена на 4 части в каждую часть доски можно положить одну из фигур. У доски должны быть методы которые помещают и удаляют фигуру с доски. Также должен быть метод который выводит информацию о всех фигурах лежащих на доске и их суммарную площадь.
5. \* Нарисуйте UML диаграмму проекта.

# Анонимные классы

## Дополнительный материал к лекции «Наследование»

Для создания единичного экземпляра класса (методы которого переопределены по сравнению с базовым) можно использовать анонимные классы.

## Определение анонимного класса

Анонимный класс – это локальный класс без имени. Можно объявить анонимный (безымянный) класс, который может расширить другой класс или реализовать (implements) интерфейс. Объявление такого класса выполняется одновременно с созданием его объекта посредством оператора new.

Анонимные классы обычно используются для реализации (переопределения) нескольких методов и создания собственных методов объекта. Так же, когда локальный класс используется всего один раз, можно применить синтаксис анонимного класса, который позволяет совместить определение и использование класса.

## Пример использования анонимного класса

```
package com.gmail.tsa;

public class ByteSequence {
    private byte[] byteArray = new byte[0];
    private int n = 0;

    private void resize(int number) {
        this.n = byteArray.length;
        byte[] temp = new byte[this.byteArray.length + number];
        System.arraycopy(byteArray, 0, temp, 0, byteArray.length);
        byteArray = temp;
    }

    public void addStringToSequence(String text) {
        byte[] byteArrayFromString = text.getBytes();
        int number = byteArrayFromString.length;
        this.resize(number);
        System.arraycopy(byteArrayFromString, 0, byteArray, n, number);
    }

    public String getString() {
        String text = new String(this.byteArray);
        return text;
    }
}
```

Опишем обычный класс. Его задача - принимать строки и хранить их в виде байтовой последовательности. Также есть возможность вернуть накопленную строку.



## Пример использования анонимного класса

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
    ByteSequence bs = new ByteSequence();
```

```
    bs.addStringToSequence("Hello");
```

```
    bs.addStringToSequence(" World");
```

```
    String text = bs.getString();
```

```
    System.out.println(text);
```

Создание обычной переменной класса и работа с ней.

```
    ByteSequence bsAnonimus = new ByteSequence() {
```

```
        @Override
```

```
        public String getString() {
```

```
            return super.getString().substring(0, super.getString().length() / 2);
```

```
        }
```

```
    };
```

```
    bsAnonimus.addStringToSequence("Hello");
```

```
    bsAnonimus.addStringToSequence(" World");
```

```
    String textOne = bsAnonimus.getString();
```

```
    System.out.println(textOne);
```

```
}
```

Использование объекта анонимного класса

Предположим, нам нужно создать один экземпляр класса ByteSequence, который возвращал бы только половину строки. Для этого можно использовать анонимный класс.

## Синтаксис создания анонимного класса

Для создания анонимного класса после вызова конструктора откройте фигурные скобки тела вашего анонимного класса



```
ByteSequence bsAnonimus = new ByteSequence() {
```

```
@Override  
public String getString() {  
    return super.getString().substring(0, super.getString().length() / 2);  
}
```

```
};
```



Переопределите требуемые методы

Внимание! Обратите внимание на то, что после закрытия тела анонимного класса нужно ставить «;»

При необходимости вы можете и не прикреплять вновь созданный объект к ссылке.

## Свойства анонимных классов

### Ограничения анонимных классов

- Конструкторы в анонимных классах ни определять, ни переопределять нельзя. Анонимный класс не может иметь конструкторов, поскольку имя конструктора должно совпадать с именем класса, а в данном случае класс не имеет имени.
- Анонимный класс может реализовать только один интерфейс.
- Анонимный класс не может ничего наследовать - (он неявный наследник базового).
- Анонимный класс не может определять статические поля, методы или классы, кроме констант `static final`.
- В анонимном классе вы не можете объявить статические инициализационные блоки.
- В анонимном классе вы не можете объявить интерфейс.

### Возможности анонимных классов (Внимание! За его пределами они будут не видны)

В анонимном классе вы можете объявить:

- Свойства;
- Дополнительные методы (даже если этих методов нет в классе родителе);
- Инициализационные блоки экземпляра;
- Локальные классы (внутренние классы);

## Пример объявления в анонимном классе свойств и методов, которых не было в базовом классе

```
ByteSequence bsAnonimusOne = new ByteSequence() {
```

```
    public String anotherText = "Hello";
```

Свойство, присущее только анонимному классу

```
    public void addIntToByteSequince(int n) {  
        String number = Integer.toString(n);  
        super.addStringToSequence(number);  
    }
```

Метод, присущий только анонимному классу

```
@Override
```

```
    public String getString() {  
        this.addIntToByteSequince(12345);  
        this.addStringToSequence(anotherText);  
        return super.getString();  
    }
```

Переопределение метода базового класса с использованием новых методов анонимного.

```
};
```

**Еще раз внимание!** Хотя метод **addIntToByteSequince** и свойство **anotherText** отмечены модификатором **public**, использовать их за пределами тела анонимного класса невозможно.

## Реализация интерфейса анонимным классом.

Анонимный класс можно использовать для «разовой» реализации интерфейса.

Чаще всего (хотя и не обязательно) в реализации опускают прикрепление объекта к ссылке. В таком случае после ключевого слова `new` следует имя реализуемого интерфейса. В теле анонимного класса должны быть методы, переопределяющие методы указанного интерфейса.

## Пример реализации интерфейса Comparator с помощью анонимного класса

```
package com.gmail.tsa;  
  
import java.util.Arrays;  
import java.util.Comparator;
```

Задача : отсортировать массив Integer  
(Это важно! С примитивными типами не  
сработает), по возрастанию модуля числа

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Integer[] array = { 3, 5, -1, 4, 1, -2, -5, 0 };  
  
        Arrays.sort(array, new Comparator<Integer>() {
```

```
            @Override  
            public int compare(Integer arg0, Integer arg1) {  
                return Math.abs(arg0) - Math.abs(arg1);  
            }  
        });
```

Реализация  
метода  
интерфейса

```
        System.out.println(Arrays.toString(array));  
    }  
}
```

За сортировку элементов массива отвечает интерфейс Comparator.  
Так как нам вряд ли еще где-нибудь понадобится такая сортировка, то  
логично ее реализовать с помощью анонимного класса.

## Краткие итоги урока

**Анонимные** классы - удобный инструмент разработки, применяемый в случае:

- Необходимости создания единственного экземпляра класса с переопределенными методами.
- Необходимости однократной реализации интерфейса.

Анонимный класс создается путем переопределения (добавления) методов базового класса. Это переопределение описывается на этапе создания объекта анонимного класса.

И хотя анонимные классы обладают немалым перечнем ограничений, они, все-таки, активно используются как удобный инструмент.

## Дополнительная литература по теме данного урока.

- Брюс Эккель Философия Java. 4-е издание. стр. 253-256
- Кей Хорстман, Гари Корнел Библиотека профессионала Java. Том 1. Основы. 9-е издание. стр. 301-304



# Перечисления

## Дополнительный материал к лекции «Наследование »

Перечисление - наследник класса Enum и является списком перечисляемых констант.

Составил: Цымбалюк А.Н.

## Основные сведения о перечислениях

Все перечисления являются наследниками (неявными) класса Enum. Однако перечисления описываются с помощью ключевого слова enum. Перечисление является списком открытых статических конечных свойств (констант).

### Пример объявления перечисления

```
package com.gmail.tsa;
```

```
public enum Colors {
```

```
RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
```

```
}
```



Список описанных констант этого перечисления.  
В данном примере это цвета радуги.

## Создание и сравнение объектов перечисления

Объект перечисления **не создается** с помощью оператора **new** . Он создается как переменная примитивного типа. Присвоить объекту можно только значение, объявленное в перечислении. По сути это означает, что может существовать только такое количество разных объектов перечисления, сколько их перечислено. Это дает возможность сравнивать объекты перечислений, используя оператор **==**.

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Colors colorOne;  
        colorOne = Colors.RED;
```

Создание объекта перечисления и присвоение ему значения.

```
        if (colorOne == Colors.RED) {  
            System.out.println("Color = RED");  
        }
```

Сравнивать объекты перечислений можно оператором **==**

```
    }  
}
```

# Конструкторы и свойства перечислений

Так как перечисления унаследованы от класса, то в нем можно объявлять конструкторы (только с модификатором `private`), собственные методы и свойства.

```
package com.gmail.tsa;
```

```
public enum Colors {  
    RED("#FF0000"), ORANGE("#FFA500"), YELLOW("#FFFF00"), GREEN("#008000"),  
    BLUE("#0000FF"), INDIGO("#4B0082"), VIOLET("#EE82EE");
```

```
private String colorCode;
```

Добавление свойства

```
private Colors(String colorCode) {  
    this.colorCode = colorCode;  
}
```

Добавление конструктора. Обратите внимание на модификатор доступа `private`

```
public String getColorCode() {  
    return colorCode;  
}  
  
public void setColorCode(String colorCode) {  
    this.colorCode = colorCode;  
}
```

Добавление методов

```
}
```

## Пример использования перечислений с конструктором и дополнительными методами

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
        Colors colorOne;  
        colorOne = Colors.RED;  
  
        if (colorOne == Colors.RED) {  
            System.out.println("Color = RED");  
        }  
  
        System.out.println(colorOne.getColorCode());  
        colorOne.setColorCode("12345");  
        System.out.println(colorOne.getColorCode());  
    }  
}
```



Использование методов, описанных в перечислении

## Методы, которые перечисления наследуют от Enum

Так как перечисления являются наследниками, Enum то от этого класса они наследуют ряд методов.

Метод	Описание
<code>static Enum valueOf(Class enumClass, String name)</code>	Возвращает перечисление класса заданного параметром <code>enumClass</code> , с именем заданным параметром <code>name</code> .
<code>static Enum valueOf( String name)</code>	Возвращает перечисление с именем заданным параметром <code>name</code> .
<code>String toString()</code>	Возвращает имя перечисляемой константы
<code>int original()</code>	Возвращает позицию данной константы в перечислении. Отчет начинается с 0.
<code>int compareTo(E other)</code>	Вернет положительное число, если текущая константа идет после той, что является параметром, отрицательное если перед и 0, если они равны.
<code>T [] values()</code>	Вернет массив всех констант данного перечисления.

# Пример использования унаследованных от Enum методов в перечислении

```
package com.gmail.tsa;
```

```
public class Main {
```

Получение экземпляра перечисления по имени константы

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

```
Colors colorTwo = Colors.valueOf(Colors.class, "ORANGE");
```

```
System.out.println(colorTwo.toString());
```

```
int position = colorTwo.ordinal();
```

Получение номера текущей константы в списке констант перечисления. Для оранжевого это 1.

```
System.out.println(position);
```

```
int comp = colorTwo.compareTo(Colors.RED);
```

Сравнение позиций констант двух перечислений.

```
System.out.println(comp);
```

```
Colors[] enumArrays = Colors.values();
```

```
for (Colors colors : enumArrays) {  
    System.out.println(colors);  
}
```

Получение массива констант перечисления и проход по ним циклом.

```
}
```

## Некоторые ограничения перечислений

На перечисления накладываются следующие ограничения:

- 1) Перечисление не может наследоваться от другого класса.
- 2) Перечисление не может быть суперклассом. Это означает, что перечисление не может быть расширено.
- 3) Перечисление может иметь только `private` конструктор



## Итоги урока

При программировании возникает необходимость ограничить множество допустимых значений для некоторого типа данных. Так, например, день недели может иметь 7 разных значений, месяц в году - 12, а время года - 4.

Для решения подобных задач предусмотрен специальный тип данных - **перечисление** (enum).

Перечисление — это список именованных констант. Описывается перечисление с помощью ключевого слова **enum**.

Создать объект перечисления с помощью ключевого слова new нельзя. Объект перечисления создается путем присвоения одной из констант этого перечисления. Так как будет создано столько экземпляров класса перечисления сколько в нем описано констант, то объекты перечисления можно сравнивать между собой оператором ==.

Неявно все перечисления являются наследниками класса Enum и реализуют интерфейсы Serializable, Comparable. Так как перечисления по сути являются классами, то в них можно описывать пользовательские члены (свойства и методы).

И хотя для перечислений существует несколько ограничений, их использование довольно распространено.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 317 — 326
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала Java . Том 1. Основы. Девятое издание. стр. 244-246

# Java OOP

(Полиморфизм и работа с исключениями)

**Полиморфизм** представляет собой концепцию, предполагающую использование единого имени (идентификатора) при обращении к объектам нескольких разных классов, при условии, что все они являются подклассами одного общего суперкласса.

Составил: Цымбалюк А.Н.

## Полиморфизм методов класса

Для каждого типа возвращаемой переменной можно реализовать свой метод, который будет отличаться типом возвращаемой переменной и списком параметров, но имя метода будет одинаковым. Такие методы называются перегруженными и относятся к «раннему» связыванию. Т.е., вариант вызванного метода определяется еще на этапе компиляции.

```
package com.gmail.tsa;
```

```
public class Multiply {
```

```
    int mul(int a, int b, int c){  
        return a*b*c;  
    }
```

```
    double mul(double a, double b, double c){  
        return a*b*c;  
    }
```

```
    String mul(String a, String b, String c){  
        return a+b+c;  
    }
```

```
}
```

Перегруженные методы



## Пример использования перегруженных методов

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
        Multiply ml= new Multiply();  
  
        System.out.println(ml.mul(1.2, 2.4, 3.5));  
        System.out.println(ml.mul(1, 2, 3));  
        System.out.println(ml.mul("ab", "cd", "ef"));  
    }  
}
```



Использование перегруженных  
методов с разными параметрами



Методы перегружать можно не только по типу параметров, но и по их количеству.

## Полиморфизм методов в подклассах (переопределение методов)

**Переопределение** метода (англ. Method overriding) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

Реализация метода в подклассе переопределяет (заменяет) его реализацию в суперклассе, описывая метод с тем же названием, что и у метода суперкласса, а также у нового метода подкласса должны быть те же параметры или сигнатура, тип возвращаемого результата, что и у метода родительского класса.

Версия метода, которая будет исполняться, определяется объектом, используемым для его вызова. Если вызов метода происходит от объекта родительского класса, то выполняется версия метода родительского класса, если же объект подкласса вызывает метод, то выполняется версия дочернего класса.

В таком случае говорят о «позднем» связывании. Т.е., вариант метода, который будет вызван, определяется уже на этапе работы программы.

## Пример полиморфизма в переопределенных методах

```
package com.gmail.tsa;
```

```
public class Multiply2 extends Multiply{
```

```
@Override
```

```
double mul(double a, double b, double c){  
    return a+b+c;  
}
```

← Метод переопределен

```
}
```

```
package com.gmail.tsa;
```

```
public class Main {
```

```
public static void main(String[] args) {  
    Multiply m1= new Multiply();  
    Multiply2 m2=new Multiply2();
```

```
System.out.println(m1.mul(2.0, 3.0, 4.0));  
System.out.println(m2.mul(2.0, 3.0, 4.0));  
m1=new Multiply2();  
System.out.println(m1.mul(2.0, 3.0, 4.0));
```

```
}
```

```
}
```

Использование переопределенных  
методов

Название методов одинаково,  
но так как они принадлежат  
разным объектам, их  
результат разный

# Использование оператора instanceof

## Объявление

Объект instanceof Имя класса

Проверяем, принадлежит ли объект указанному классу. Вернет true, если да и false, если наоборот.

## Пример использования

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
        Multiply m1= new Multiply();  
        Multiply2 m2=new Multiply2();  
  
        System.out.println((m1 instanceof Multiply2));  
    }  
}
```

Проверка принадлежит ли m1 классу Multiply



## Исключения

**Исключение** в Java представляет собой объект, который описывает исключительную (ошибочную) ситуацию, возникающую при выполнении программного кода.

Обработка исключений осуществляется с помощью ключевых слов :

- try
- catch
- throw
- throws
- finally

## Блок try.. catch.. finally..

Блок операторов, которые нужно проверить на наличие исключений, помещают внутрь блока **try**

```
try{  
    Операторы...  
}
```

Для обработки исключения используется оператор **catch**

```
catch(Тип исключения e){  
    Операторы выполняемые при перехвате исключения  
}
```

```
finally {  
    - блок операторов, которые выполняться после блока try и catch, в  
      независимости от того, какой из них отработал.  
}
```

## Пример обработки исключения

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        int a=3;
```

```
        int b=0;
```

Намеренный вызов ошибки (деление на ноль)

```
        try{
```

```
            System.out.println(a/b);
```

```
        }
```

```
        catch(ArithmeticException e){  
            System.out.println("Division by zero");  
        }
```

Перехват исключения

```
        finally{
```

```
            System.out.println("But the program is still running");
```

```
        }
```

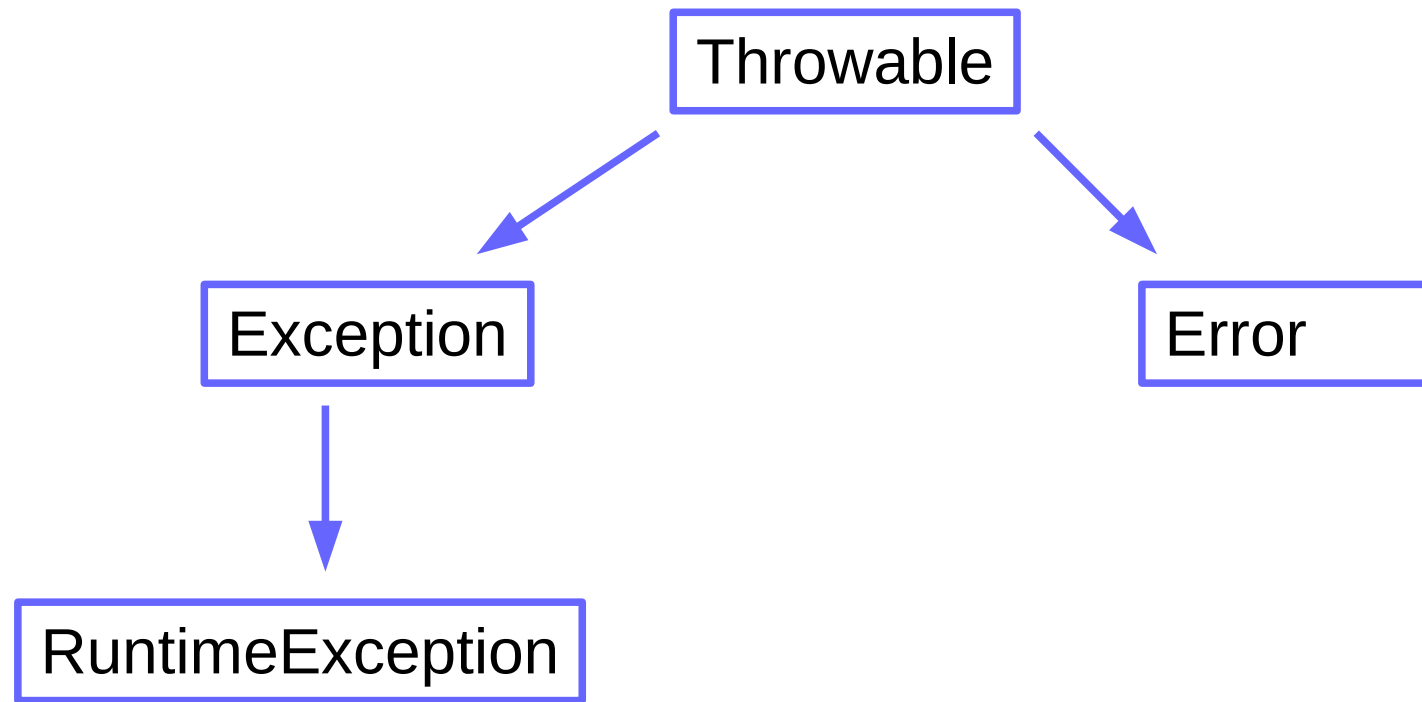
```
        System.out.println("End");
```

```
    }
```

```
}
```

Участок кода, который выполнится в любом случае

# Иерархия исключений



`RuntimeException` - обрабатываются  
`Error` - обычно не обрабатываются

## Пример использования перехвата исключений (проверка ввода числа пользователем)

```
package com.gmail.tsa;

import javax.swing.JOptionPane;

public class Main {

    public static void main(String[] args) {
        double a;

        for(;;){

            try{
                a=Double.valueOf(JOptionPane.showInputDialog("Input double number"));
                break;
            }
            catch(NumberFormatException e){
                JOptionPane.showMessageDialog(null,"Error number format");
            }
        }
        System.out.println(a);
    }
}
```

## Вложенность операторов try и catch

Для одного блока try может быть написано несколько операторов catch с целью перехвата ошибок разного типа.

Также возможно использование нескольких, вложенных друг в друга, блоков try. В таком случае сначала проверяется блок catch вложенного блока try, а потом внешнего.

**Внимание!** Если исключение в вышестоящем блоке catch стоит выше по иерархии исключения в нижестоящем блоке, то нижестоящий блок никогда не выполнится.

# Оператор throw

**throw** - используется для явной передачи исключения

Общая форма оператора

**throw** объект\_класса\_Throwable

Выполнение последующих операторов прекратится сразу после оператора **throw**

Распространенной практикой является генерация исключения класса **IllegalArgumentException**. Это исключение следует генерировать, если параметр метода корректен с точки зрения типа, но не корректен с точки зрения значения.

```
public double calculateArea(double r) {
```

```
    if (r < 0) {
```

← Если радиус круга меньше нуля

```
        throw new IllegalArgumentException("Negative radius");
```

```
    }  
    return Math.PI * r * r;
```

```
}
```

← То генерируем исключение, так как этот параметр некорректен по значению

## Пример генерации исключения

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        try{  
            for(int i=0;i<10;i++){
```

Генерация нового исключения



```
                if(i==5) throw new ArithmeticException();
```

```
            }
```

```
        }
```

```
        catch(ArithmeticException e){  
            System.out.println("Interception thrown");
```

Перехват исключения



```
        }
```

```
        finally{  
            System.out.println("finally block");
```

```
        }
```

```
        System.out.println("End");
```

```
    }
```

```
}
```



## Оператор throws

Если метод может вызвать исключение, но сам не обрабатывает его, то к описанию метода следует добавить оператор **throws**.

В таком случае общая форма объявления метода выглядит как

Тип имя\_метода (параметры) throws список\_исключений

## Пример использования throws

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        try{
```

```
            System.out.println(sum(3,4));
```

```
            System.out.println(sum(5,6));
```

```
        }
```

```
        catch(IllegalAccessException e){
```

```
            System.out.println("Interception thrown");
```

```
        }
```

```
    }
```

Метод создает исключение, но не обрабатывает



```
    static int sum(int a, int b) throws IllegalAccessException {
```

```
        if (a==5) throw new IllegalAccessException();
```

```
        return a+b;
```

```
    }
```



Генерация исключения

```
}
```

Перехват исключения

## Проверяемые исключения класса RuntimeException

Исключение	Описание
ClassNotFoundException	Класс не найден
CloneNotSupportedException	Попытка клонировать объект, который не реализует интерфейс Cloneable
IllegalAccessException	Доступ к классу не разрешен
InstantiationException	Попытка создать объект абстрактного класса или интерфейса
InterruptedException	Один поток прерван другим потоком
NoSuchFieldException	Запрошенное свойство (поле) не существует
NoSuchMethodException	Запрошенный метод не существует
ReflectiveOperationException	Исключения рефлексии

Исключения проверяемого типа должны быть описаны в операторе throws

## Создание собственных подклассов исключений

Для создания собственного подкласса исключений нужно:

- 1) Создать класс который будет наследником класса Exception.
- 2) Переопределить или дополнить существующие методы.

Пример создания подкласса исключения

```
package com.gmail.tsa;  
  
public class MyException extends Exception{  
    @Override  
    public String getMessage(){  
        return " Subclass exceptions created";  
    }  
}
```

Переопределение метода  
Класса Exception

# Использование пользовательского исключения

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        try{
```

```
            System.out.println(sum(3,4));
```

```
            System.out.println(sum(5,6));
```

```
        }
```

```
        catch(MyException e){
```

```
            System.out.println(e.getMessage());
```

```
        }
```

```
    }
```

```
    static int sum(int a,int b) throws MyException {
```


```
        if (a==5) throw new MyException();
```

```
        return a+b;
```


```
    }
```

```
}
```


Обработка пользовательского  
исключения



Метод, передающий  
пользовательское исключение



Создание пользовательского  
исключения



## Использование мультиобработчика исключений

Начиная с JDK 7 появилось новое средство - мультиобработчик исключений.

Суть его в том, что в блоке catch можно перечислять исключения, используя оператор `|` - «или».

Пример:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
```

## Краткие итоги урока

**Полиморфизм** — инструмент, направленный на вариативность (возможность выполнения разных действий по одному индикатору). В java полиморфизм реализован в виде полиморфизма методов. Т.е., несколько методов могут иметь одно название.

К полиморфизму можно отнести два случая:

**Перегрузка** — методы класса могут иметь одно название, но разные списки параметров, где вариант вызванного метода будет зависеть от фактического набора параметров.

**Переопределение** — метод подкласса обладает абсолютно одинаковой сигнатурой с методом суперкласса. Вариант вызванного метода определится типом объекта.

**Исключение** - механизм, направленный на написание безопасного (с точки зрения надежности выполнения) кода. В Java реализован механизм как перехвата и обработки исключений, так и генерации и создания собственного типа исключений.

Обработка исключений выполняется операторами try и catch.

Генерация исключения throw.

## Дополнительная литература по теме данного урока.

- Брюс Эккель Философия Java. 4-е издание. стр. 310-351
- Герберт Шилд. Java. Полное руководство 8-е издание. стр. 239 - 257



## Домашнее задание

- 1) Создайте класс, описывающий человека (создайте метод, выводящий информацию о человеке).
- 2) На его основе создайте класс студент (переопределите метод вывода информации).
- 3) Создайте класс Группа, который содержит массив из 10 объектов класса Студент. Реализуйте методы добавления, удаления студента и метод поиска студента по фамилии. В случае попытки добавления 11 студента, создайте собственное исключение и обработайте его. Определите метод toString() для группы так, что бы он выводил список студентов в алфавитном порядке.
- 4) \* Нарисуйте UML диаграмму проекта.

# Механизм рефлексии

## Дополнительный материал к лекции «Полиморфизм»

Рефлексия - это механизм исследования данных о программе во время ее выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов. Можно также выполнять операции над полями и методами, которые исследуются.

**Составил: Цымбалюк А.Н.**

## Возможности интерфейса Java Reflection API:

- Определение класса объекта.
- Получение информации о модификаторах класса, полях, методах, конструкторах и суперклассах.
- Определение констант и методов, принадлежащих интерфейсу.
- Создание экземпляра класса, имя которого неизвестно до момента выполнения программы.
- Получение и установка значения свойства объекта.
- Вызов методов объекта.
- Создание нового массива, размер и тип компонентов которого, неизвестны до момента выполнения программ.

Интерфейс Java Reflection API состоит из классов пакетов `java.lang` и `java.lang.reflect`.

Для начала работы с Reflection API требуется получить экземпляр класса Class

Выполнить это можно несколькими способами:

- Использовать свойство `class` для существующего класса;
- Вызвать метод `getClass()` у существующего объекта;
- Вызвать метод `Class.forName("Class_name");`

```
package com.gmail.tsa;  
  
import java.util.Scanner;
```

## Пример получения объектов Class

```
public class Main {
```

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

```
        Scanner sc = new Scanner(System.in);
```

```
        Class<?> classTwo = Scanner.class;
```

Вызов для класса

```
        Class<?> classOne = sc.getClass();
```

Вызов для объекта

```
        Class<?> classThree = null;
```

```
        try {
```

```
            classThree = Class.forName("java.util.Scanner");
```

```
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();
```

```
        }
```

```
        System.out.println(classOne);
```

```
        System.out.println(classTwo);
```

```
        System.out.println(classThree);
```

```
        sc.close();
```

```
    }
```

```
}
```

Получение по имени

Внимание! При получении по имени, важно указывать полное имя класса, включая пакеты, где он лежит.

После получения переменной Class можно провести ее исследование

## Таблица методов для исследования переменной типа Class

Название метода	Тип возвращаемого значения	Описание
getSuperclass()	Class<?>	Вернет Class суперкласса данного класса.
getInterfaces()	Class<?>[]	Вернет массив Class - интерфейсов, которые реализует данный класс.
getModifiers();	int	Вернет int, в котором зашифрованы модификаторы данного класса.
getFields()	Field[]	Массив Field — т. е. свойств этого класса.
getConstructors()	Constructor[]	Массив конструкторов данного класса.
getMethods()	Method[]	Массив методов данного класса.

## Пример использования Class для исследования класса

```
package com.gmail.tsa;
```

```
import java.io.File;
```

```
public class Main {
```

Получение Class суперкласса текущего класса

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

```
        File file = new File("a.txt");
```

Создание экземпляра класса

```
        Class <?> fileClass = file.getClass();
```

Получение Class объекта

```
        Class<?> superFileClass = fileClass.getSuperclass();
```

```
        System.out.println(superFileClass.getName());
```

```
        Class <?>[] implementsInterface = fileClass.getInterfaces();
```

```
        for (Class <?> inter : implementsInterface) {  
            System.out.println(inter.toString());  
        }
```

```
    }
```

Получение массива интерфейсов, которые реализует данный класс

## Получение и определение модификаторов

Модификатор класса, свойства, метода можно получить, вызвав для них метод `getModifiers()`. Этот метод вернет переменную типа `int`, в которой в виде байтовой последовательности записаны значения модификаторов. Последние 11 бит этого числа кодируют эти модификаторы в соответствии с приведенной таблицей.

бит	11	10	9	8	7	6	5	4	3	2	1	0
целое значение	2048	1024	512	256	128	64	32	16	8	4	2	1
модификатор	<code>strictfp</code>	<code>abstract</code>	<code>interface</code>	<code>native</code>	<code>transient</code>	<code>volatile</code>	<code>synchronized</code>	<code>final</code>	<code>static</code>	<code>protected</code>	<code>private</code>	<code>public</code>

Для облегчения расшифровки значений модификаторов существуют статические методы `isPublic(int mod)`, `isPrivate(int mod)` и т. д. служебного класса `Modifier`.



## Пример получения и расшифровки модификаторов класса

```
package com.gmail.tsa;  
  
import java.io.File;  
import java.lang.reflect.Modifier;
```

```
public class Main {
```

Получение модификаторов класса в виде int

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

```
        File file = new File("a.txt");  
        Class <?> fileClass = file.getClass();
```

```
        int mod = fileClass.getModifiers();
```

```
        System.out.println(Integer.toBinaryString(mod));
```

```
        System.out.println("Public class " + Modifier.isPublic(mod));
```

```
        System.out.println("Private class " + Modifier.isPrivate(mod));
```

```
        System.out.println("Abstarct class " + Modifier.isAbstract(mod));
```

```
    }  
}
```

Использование статических методов класса Modifier для расшифровки результатов

## Получение свойств класса

После получение экземпляра Class можно получить список как открытых свойств класса, так и полный список (включая protected,private). За отображение свойств класса отвечают экземпляры класса Field.

Выполнить это можно несколькими способами:

- Метод `getFields()` вернет массив открытых свойств класса;
- Метод `getField(«Field name»)` вернет одно открытое свойство класса по его имени;
- Метод `getDeclaredFields()` вернет массив всех свойств класса;
- Метод `getDeclaredField(«Field name»)` вернет одно свойство класса по его имени;

**Внимание! Унаследованные свойства так получить не получится! Относится только к `getDeclaredFields()` и `getDeclaredField(«Field name»)`**

Для использования необходимо импортировать  
`java.lang.reflect.Field;`

Для дальнейшего рассмотрения введем вспомогательный класс, чтобы исследовать его методами рефлексии

```
package com.gmail.tsa;
```

```
import java.io.File;  
import java.io.FileInputStream;  
import java.io.IOException;
```

```
public class FileWorker {  
    public String fileName;  
    protected long size;  
    private boolean isByteFile = true;  
    private File file;
```

Свойства класса для исследования

```
    public FileWorker(String fileName) {  
        super();  
        this.fileName = fileName;  
        this.size = getFileSize(fileName);  
    }
```

Конструктор

```
    private final long getFileSize(String fileName) {  
        this.file = new File(fileName);  
        return file.length();  
    }
```

```
    public void setByteFile(boolean isByteFile) {  
        this.isByteFile = isByteFile;  
    }
```

```
    public byte[] getByteFromFile() {  
        if (!isByteFile) {  
            throw new IllegalArgumentException("The symbolyc file");  
        }  
        byte[] byteArray = new byte[(int) this.file.length()];  
        try (FileInputStream fis = new FileInputStream(this.file)) {  
            fis.read(byteArray);  
        } catch (IOException e) {  
            System.out.println(e);  
        }  
        return byteArray;  
    }
```

Вспомогательные методы  
класса

```
}
```

## Пример получения свойств класса

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Field;
```

```
public class Main {
```

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

```
        FileWorker fw = new FileWorker("a.txt");
```

Создание экземпляра класса

```
        Class <?> cl = fw.getClass();
```

Получение Class на основе объекта

```
        Field[] fields = cl.getDeclaredFields();
```

Получение массива свойств

```
        System.out.println("All fields in class " + cl.getName() + "\n");
```

```
        for (Field field : fields) {
```

```
            Class <?> fieldClass = field.getType();
```

Получения Class свойства

```
            System.out.println(field.getName() + " - " + fieldClass.getName());
```

```
        }
```

```
        System.out.println("\nOne field in class " + cl.getName() + "\n");
```

```
        Field fieldOne = null;
```

Получение Field по имени

```
        try {
```

```
            fieldOne = cl.getDeclaredField("isByteFile");
```

```
            System.out.println(fieldOne.getType() + " - " + fieldOne.getName());
```

```
        } catch (NoSuchFieldException | SecurityException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

## Получение значений свойств объекта средствами рефлексии

После получения объекта Field можно считать значение соответствующего свойства класса.

В случае **открытых** свойств класса для **считывания** используются следующие методы:

Для считывания значений примитивных типов:

- getInt(obj), getLong(obj), getDouble(obj), getFloat(obj), getChar(obj) и т. д.

Для считывания ссылочных типов данных используется метод

- get(obj)

В случае **закрытых** свойств класса для **считывания** нужно

- 1) Открыть доступ к свойству, используя метод setAccessible(true);
- 2) Считать его как открытое.

obj — это объект, значение свойства которого нужно получить.

## Пример получения значения открытого свойства класса

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Field;
```

```
public class Main {
```

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

Создание экземпляра класса

```
        FileWorker fw = new FileWorker("a.txt");
```

```
        Class <?> cl = fw.getClass();
```

Получение Class на основе объекта

```
        Field fieldOne = null;
```

```
        String text = "";
```

```
        try {
```

Получение Field по имени

```
            fieldOne = cl.getDeclaredField("fileName");
```

```
            text = (String) fieldOne.get(fw);
```

Получение значения свойства

```
        } catch (NoSuchFieldException | SecurityException  
                | IllegalArgumentException | IllegalAccessException e) {  
            e.printStackTrace();  
        }
```

```
        System.out.println(text);
```

```
    }
```

```
}
```

## Пример получения значения закрытого свойства класса

```
package com.gmail.tsa;

import java.lang.reflect.Field;

public class Main {

    public static void main(String[] args) {

        FileWorker fw = new FileWorker("a.txt");
        Class<?> cl = fw.getClass();
        Field fieldOne = null;
        boolean isByteFile = false;
        try {
            fieldOne = cl.getDeclaredField("isByteFile");
            fieldOne.setAccessible(true);
            isByteFile = fieldOne.getBoolean(fw);
        } catch (NoSuchFieldException | SecurityException
            | IllegalArgumentException | IllegalAccessException e) {
            e.printStackTrace();
        }
        System.out.println(isByteFile);
    }
}
```

Создание экземпляра класса

Получение Class на основе объекта

Получение Field по имени

Считывание значения

Установка разрешения на чтение private свойства

## Установка значений свойств объекта средствами рефлексии

После получения объекта Field можно установить значение соответствующего свойства класса.

В случае **открытых** свойств класса для **установки** используются следующие методы:

Для установки значений примитивных типов:

- `setInt(obj, value)`, `setLong(obj, value)`, `setDouble(obj, value)`, `setFloat(obj, value)`, `setChar(obj, value)` и т. д.

Для установки ссылочных типов данных используется метод

- `set(obj, value)`

В случае **закрытых** свойств класса для **установки** нужно:

- 1) Открыть доступ к свойству, используя метод `setAccessible(true)`;
- 2) Установить его как открытое.

`obj` — это объект, значение свойства которого нужно получить.

`value` — новое значение.



## Пример установки значения открытого свойства класса

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Field;
```

```
public class Main {
```

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

```
        FileWorker fw = new FileWorker("a.txt");
```

```
        Class<?> fileClass = fw.getClass();
```

```
        Field fieldOne;
```

```
        try {
```

```
            fieldOne = fileClass.getDeclaredField("size");
```

```
            fieldOne.setLong(fw, 912345L);
```

```
        } catch (NoSuchFieldException | SecurityException |  
IllegalArgumentExcepTion | IllegalAccessException e) {  
            e.printStackTrace();  
        }
```

```
        System.out.println(fw.size);
```

```
    }  
}
```

Создание экземпляра класса

Получение Class на основе  
объекта

Получение Field по имени

Установка значения свойства для объекта

## Подробнее об установке открытых свойств класса

Для какого объекта  
устанавливается свойство

Новое значение устанавливаемого  
свойства

Получение Field по имени

```
try {  
    fieldOne = fileClass.getDeclaredField("size");  
    fieldOne.setLong(fw, 912345L);  
} catch (NoSuchFieldException | SecurityException |  
        IllegalArgumentException |  
        IllegalAccessException e) {  
    e.printStackTrace();  
}
```

Исключения, связанные с получением  
Field и установкой свойства для объекта

Установка значения свойства для объекта

## Пример установки значения закрытого свойства класса

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Field;
```

```
public class Main {
```

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

```
        FileWorker fw = new FileWorker("a.txt");  
        Class<?> fileClass = fw.getClass();
```

```
        Field fieldOne;
```

```
        try {  
            fieldOne = fileClass.getDeclaredField("isByteFile");
```

```
            fieldOne.setAccessible(true);
```

```
            fieldOne.setBoolean(fw, false);
```

```
        } catch (NoSuchFieldException | SecurityException |  
                IllegalArgumentException | IllegalAccessException e) {  
            e.printStackTrace();  
        }
```

```
        byte[] mass = fw.getBytesFromFile();
```

```
    }
```

```
}
```

Создание экземпляра класса

Получение Class на  
основе объекта

Получение Field по имени

Установка доступа

Установка значения свойства

## Получение списка методов класса средствами рефлексии

За отображения метода класса отвечает `java.lang.reflect.Method`.  
Для его получения необходимо:

В случае использования **открытых** методов

- Для получения массива методов вызвать `getMethods()` для объекта `Class`;
- Для получения метода по имени и списку параметров вызвать `getMethod(String "methodName", Class<?>[] paramTypes);`

В случае использования **закрытых** методов

- Для получения массива методов вызвать `getDeclaredMethods()` для объекта `Class`;
- Для получения метода по имени и списку параметров вызвать `getDeclaredMethod(String "methodName", Class<?>[] paramTypes);`

Как и в случае со свойствами `getDeclaredMethods()` не вернет унаследованные методы.

## Пример получения массива Method на основе класса

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Method;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        FileWorker fw = new FileWorker("a.txt");  
        Class<?> classOne = fw.getClass();
```

Создание экземпляра класса

Получение Class на основе объекта

```
        Method[] methodList = classOne.getDeclaredMethods();
```

Получение массива Method

```
        for (Method method : methodList) {  
            System.out.println("Method name - " + method.getName());
```

```
            System.out.println("Return type - " +  
            method.getReturnType().getName());
```

```
            Class<?>[] parametresType = method.getParameterTypes();  
            System.out.println("Parametr list:");
```

```
                for (Class<?> class1 : parametresType) {  
                    System.out.println("Type - " + class1.getName());
```

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

```
        }
```

```
    }
```

```
}
```

Получение параметров Method

## Пример получения Method на основе имени и списка параметров

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Method;
```

```
public class Main {
```

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

```
        FileWorker fw = new FileWorker("a.txt");
```

```
        Class<?> classOne = fw.getClass();
```

```
        Method method = null;
```

```
        try {
```

```
            method = classOne.getDeclaredMethod("getFileSize", String.class);
```

```
        } catch (NoSuchMethodException | SecurityException e) {  
            e.printStackTrace();  
        }
```

```
        System.out.println("Method name - " + method.getName());
```

```
        System.out.println("Return type - " +  
            method.getReturnType().getName());
```

```
    }  
}
```

Создание экземпляра класса

Получение Class на основе объекта

Объявление переменной типа Method

Получение Method по имени и типу параметров

Получение параметров Method

После получения Method вы можете, как исследовать его так и ВЫПОЛНИТЬ.

Название метода	Описание
<code>int getModifiers()</code>	Вернет модификатор метода.
<code>int getParameterCount()</code>	Вернет количество формальных параметров.
<code>Class&lt;?&gt;[] getParameterTypes()</code>	Вернет массив типов параметров метода.
<code>Class&lt;?&gt; getReturnType()</code>	Вернет тип, возвращаемого методом, значения.
<code>Class&lt;?&gt;[] getExceptionTypes()</code>	Вернет массив типов генерируемых исключений.
<code>String getName()</code>	Вернет название метода в виде строки.
<code>Class&lt;?&gt; getDeclaringClass()</code>	Вернет тип класса или интерфейса, где метод был объявлен.
<code>Object invoke(Object obj, Object... args)</code>	Выполнит метод с указанным списком параметров для переменной нужного класса.

Внимание! Список методов не полный. Есть еще серия методов, касающаяся работы с аннотациями. О них в следующем дополнении.

## Пример выполнения метода средствами рефлексии

```
package com.gmail.tsa;
```

```
import java.lang.reflect.InvocationTargetException;
```

```
import java.lang.reflect.Method;
```

```
public class Main {
```

Создание экземпляра класса

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

```
    FileWorker fw = new FileWorker("a.txt");
```

```
    Class<?> classOne = fw.getClass();
```

Получение Class на основе объекта

```
    long result = -1;
```

Объявление переменной типа Method

```
    Method method = null;
```

```
    try {
```

Получение Method по имени и типу параметров

```
        method = classOne.getDeclaredMethod("getFileSize", String.class);
```

```
        method.setAccessible(true);
```

Так как метод private, открываем доступ

```
        result = (Long) method.invoke(fw, "a.txt");
```

Выполняем метод

```
    } catch (NoSuchMethodException | SecurityException | IllegalAccessException |  
            IllegalArgumentException | InvocationTargetException e) {  
        e.printStackTrace();  
    }
```

```
    System.out.println(result);
```

```
}
```



## Процедура выполнения метода подробнее

```
long result = -1;
```

```
Method method = null;  
try {
```

Получаем имя метода, используя его имя, и то, что его параметром является строка

```
method = classOne.getDeclaredMethod("getFileSize", String.class);
```

```
method.setAccessible(true);
```

```
result = (Long) method.invoke(fw, "a.txt");
```

```
} catch (NoSuchMethodException | SecurityException |  
        IllegalAccessException | IllegalArgumentException  
        | InvocationTargetException e) {  
    e.printStackTrace();  
}
```

Invoke возвращает Object, в то время как getFileSize long, поэтому приводим к нужному типу

Указываем объект, чей метод мы хотим выполнить. Для статического укажите null. В данном случае для объекта fw

Параметр, с которым этот метод будет выполнен.

Установка разрешения выполнения private метода, для открытых методов не нужна

## Получение и работа с конструкторами класса

За отображения конструктора класса отвечает `java.lang.reflect.Constructor`

Для его получения необходимо:

В случае использования **открытых** конструкторов

- Для получения массива конструкторов вызвать `getConstructors()` для объекта `Class`;
- Для получения метода по списку параметров вызвать `getConstructor(Class<?>[] paramTypes);`

В случае использования **закрытых** конструкторов

- Для получения массива конструкторов вызвать `getDeclaredConstructors()` для объекта `Class`;
- Для получения конструктора по списку параметров вызвать `getDeclaredConstructor(Class<?>[] paramTypes);`

После получения `Constructor` их можно исследовать по аналогии с обычными методами и использовать для динамического создания объектов.

# Пример получения и исследования свойств конструктора класса

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Constructor;
```

```
public class Main {
```

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

```
        FileWorker fw = new FileWorker("a.txt");
```

```
        Class<?> classOne = fw.getClass();
```

```
        Constructor<?>[] constructorList = classOne.getDeclaredConstructors();
```

```
        for (Constructor<?> constructor : constructorList) {
```

```
            System.out.println(constructor.getName());
```

```
            System.out.println(constructor.getParameterCount());
```

```
            Class<?>[] constructorParameters = constructor.getParameterTypes();
```

```
            for (Class<?> class1 : constructorParameters) {
```

```
                System.out.println(class1.getName());
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

Создание экземпляра класса

Получение Class на основе объекта

Получение массива Constructor

Исследование Constructor по аналогии с методами

## Пример создания объекта на основе Constructor

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Constructor;
```

```
import java.lang.reflect.InvocationTargetException;
```

```
public class Main {
```

Получение Class на основе имени класса

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

```
        Class<?> classOne = FileWorker.class;
```

```
        FileWorker fw = null;
```

Объявление переменной типа Constructor

```
        Constructor<?> constructor;
```

```
        try {
```

Получение конструктора по списку параметров

```
            constructor = classOne.getConstructor(String.class);
```

```
            fw = (FileWorker) constructor.newInstance("a.txt");
```

```
        } catch (NoSuchMethodException | SecurityException |  
                InstantiationException | IllegalAccessException  
                | IllegalArgumentException | InvocationTargetException e) {  
            e.printStackTrace();
```

```
        }  
        System.out.println(fw.size);
```

Создание нового объекта на основе Constructor

```
    }  
}
```

## Динамическое создание объектов подробнее

```
Constructor<?> constructor;
```

```
try {
```

Получаем конструктор по списку параметров. (В данном случае есть только один конструктор с параметром String).

```
    constructor = class0ne.getConstructor(String.class);
```

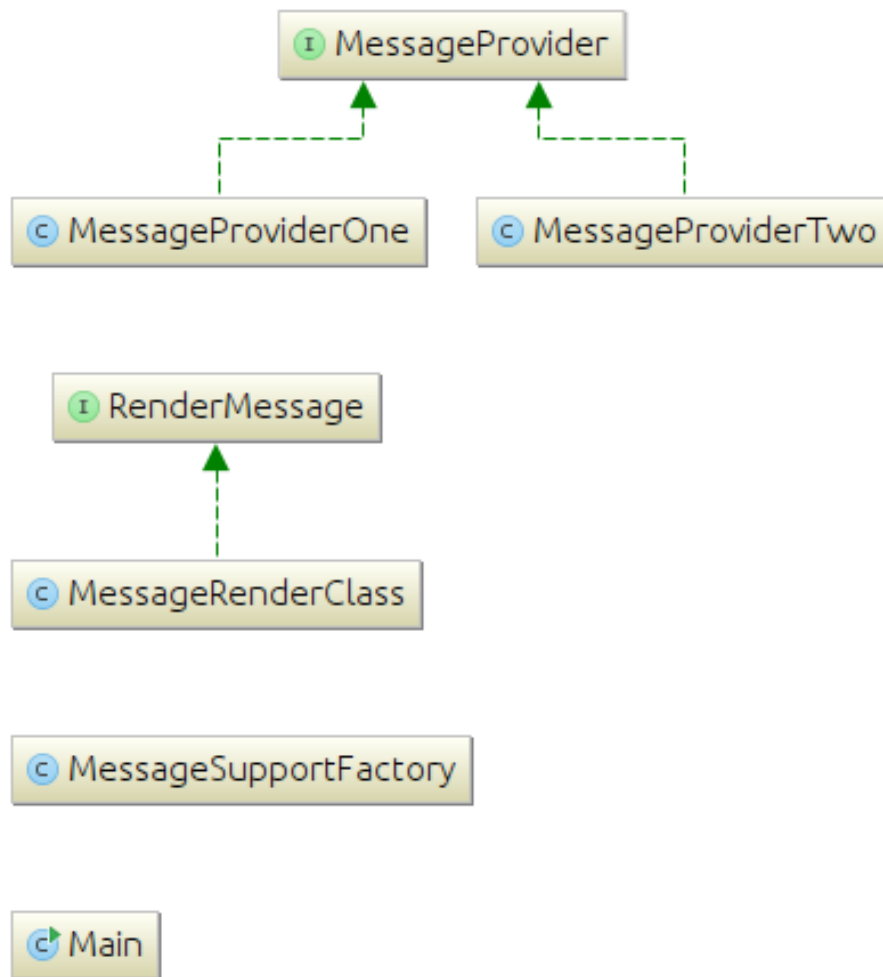
```
    fw = (FileWorker) constructor.newInstance("a.txt");
```

```
} catch (NoSuchMethodException | SecurityException |  
    InstantiationException | IllegalAccessException  
    | IllegalArgumentException | InvocationTargetException e) {  
    e.printStackTrace();  
}
```

Нужно динамически создать экземпляр класса на основе конструктора. (Так как метод `newInstance(Object... obj)` вернет переменную типа `Object`, то его необходимо привести к результирующему типу).

Метод `newInstance(Object... obj)` используется для создания объекта на основе полученного конструктора, где параметром этого метода должны быть параметры, совпадающие с параметрами конструктора.

## Пример возможного применения рефлексии



Задача: с помощью рефлексии реализовать возможность конкурирования версии загружаемого класса с помощью внешнего класса.

Есть два интерфейса: (`MessageProvider` и `Render Message`) для `MessageProvider` есть две реализации, и какая реализация будет использована, определится во внешнем файле.

Конфигурация осуществляется классом `MessageSupportFactory`.

UML диаграмма проекта изображена слева.

## Интерфейсы - проекты

```
package com.gmail.tsa;  
  
public interface MessageProvider {  
    public String getMessage();  
}
```

```
package com.gmail.tsa;  
  
public interface RenderMessage {  
    public void renderMessage();  
  
    public void setMessageProvider(MessageProvider mesprov);  
  
    public MessageProvider getMessageProvider();  
}
```

Как видно из приведенного кода, MessageProvider просто возвращает строку текста. RenderMessage рисует ее, при этом он использует MessageProvider для ее получения.

## Две реализации MessageProvider

```
package com.gmail.tsa;

public class MessageProviderOne implements MessageProvider {

    @Override
    public String getMessage() {
        return "Message One";
    }

}
```

```
package com.gmail.tsa;

public class MessageProviderTwo implements MessageProvider {

    @Override
    public String getMessage() {
        return "Message Two";
    }

}
```



## Реализация RenderMessage

```
package com.gmail.tsa;
```

```
public class MessageRenderClass implements RenderMessage {  
    private MessageProvider mesprov = null;  
  
    @Override  
    public void renderMessage() {  
        if (mesprov == null) {  
            throw new IllegalArgumentException("Set message provider");  
        }  
        System.out.println(mesprov.getMessage());  
    }  
  
    @Override  
    public void setMessageProvider(MessageProvider mesprov) {  
        this.mesprov = mesprov;  
    }  
  
    @Override  
    public MessageProvider getMessageProvider() {  
        return this.mesprov;  
    }  
}
```

Как вы видите, для корректной работы этого класса нужен экземпляр класса, реализующий MessageProvider

```
package com.gmail.tsa;
```

```
import java.io.BufferedReader;  
import java.io.File;  
import java.io.FileReader;  
import java.io.IOException;
```

```
public class MessageSupportFactory {
```

```
    private MessageProvider mespro = null;  
    private RenderMessage renmes = null;
```

```
    public MessageSupportFactory(File file) {  
        super();
```

```
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {  
            mespro = (MessageProvider) Class.forName(br.readLine()).newInstance();  
            renmes = (RenderMessage) Class.forName(br.readLine()).newInstance();
```

```
        } catch (IOException | InstantiationException | IllegalAccessException |  
                ClassNotFoundException e) {  
            System.out.println(e);  
        }  
    }
```

```
    public MessageProvider getMespro() {  
        return mespro;
```

```
    }  
    public RenderMessage getRenmes() {  
        return renmes;
```

```
    }
```

Класс, считывающий из файла имена классов, и загружающий их

Вычитка данных из файла

Получение экземпляра класса на основе его имени, полученного из файла

Методы для возврата полученной реализации интерфейса

## Главный класс проекта

```
package com.gmail.tsa;
```

```
import java.io.File;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        MessageSupportFactory msf = new MessageSupportFactory(new  
            File("config.txt"));
```

```
        MessageProvider mprov = msf.getMespro();  
        RenderMessage renMes = msf.getRenmes();
```

```
        renMes.setMessageProvider(mprov);  
        renMes.renderMessage();
```

```
    }
```

```
}
```

И их использование

Создание класса, который загрузит требуемую реализацию интерфейса на основе конфигурационного файла



Получение реализаций интерфейсов



```
com.gmail.tsa.MessageProviderOne  
com.gmail.tsa.MessageRenderClass
```



Содержимое файла config.txt

## Рефлексия и обобщенные классы

Механизм рефлексии можно использовать для ответа на такие вопросы:

- Имеет ли обобщенный метод обобщенный параметр T;
- Имеет ли параметр типа ограниченный подтип, который сам является обобщенным;
- Имеет ли ограничивающий тип подставляемый параметр;
- Имеет ли подставляемый параметр ограниченный супертип;
- Имеет ли обобщенный метод обобщенный массив в качестве параметра;

Для получения объявления обобщенных типов используют интерфейс Type. Этот интерфейс имеет указанные подинтерфейсы и классы его реализующие

- Class — описывает конкретные типы;
- TypeVariable — интерфейс описывает переменные типа;
- WildcardType — интерфейс описывает подстановки;
- ParameterizedType — интерфейс описывает обобщенный класс или интерфейс;
- GenericArrayType — интерфейс описывает обобщенные массивы ;

## Методы классов и интерфейсов механизма рефлексии для работы с обобщенными типами

`java.lang.Class<T>`

Метод	Описание
<code>TypeVariable[] getTypeParameters()</code>	Получает переменные обобщенного типа, если класс был обобщенный или массив нулевой длины, если нет.
<code>Type getGenericSuperclass()</code>	Получает обобщенный тип суперкласса, который был объявлен для этого типа, или же null, если это Object.
<code>Type [] getGenericInterfaces()</code>	Обобщенные типы интерфейсов, которые были объявлены для этого типа, в порядке объявления, или массив нулевой длины, если данный тип не реализует интерфейсы.

## Методы классов и интерфейсов механизма рефлексии для работы с обобщенными типами

`java.lang.reflect.Method`

Метод	Описание
<code>TypeVaribale[] getTypeParameters()</code>	Получает переменные обобщенного типа, если это метод был обобщенный, или массив нулевой длины, если нет.
<code>Type getGenericReturnType()</code>	Получает обобщенный возвращаемый тип, с которым объявлен данный метод.
<code>Type [] getGenericParameterTypes()</code>	Получает обобщенные типы параметров, с которыми объявлен данный метод. Если у метода отсутствуют параметры, возвращает массив нулевой длины.

## Методы классов и интерфейсов механизма рефлексии для работы с обобщенными типами

`java.lang.reflect.TypeVariable`

Метод	Описание
<code>String getName()</code>	Получает имя переменной типа.
<code>Type [] getBounds()</code>	Получает ограничения на подкласс для данной переменной типа, или массив нулевой длины, если их нет.

## Методы классов и интерфейсов механизма рефлексии для работы с обобщенными типами

`java.lang.reflect.WildcardType`

Метод	Описание
<code>Type[] getUpperBounds()</code>	Получает для данной переменной типа ограничения на подкласс, определяемые оператором <code>extends</code> , или массив нулевой длины, если ограничения отсутствуют.
<code>Type[] getUpperBounds()</code>	Получает для данной переменной типа ограничения на суперкласс, определяемые оператором <code>super</code> , или массив нулевой длины, если ограничения отсутствуют.



## Методы классов и интерфейсов механизма рефлексии для работы с обобщенными типами

`java.lang.reflect.ParameterizedType`

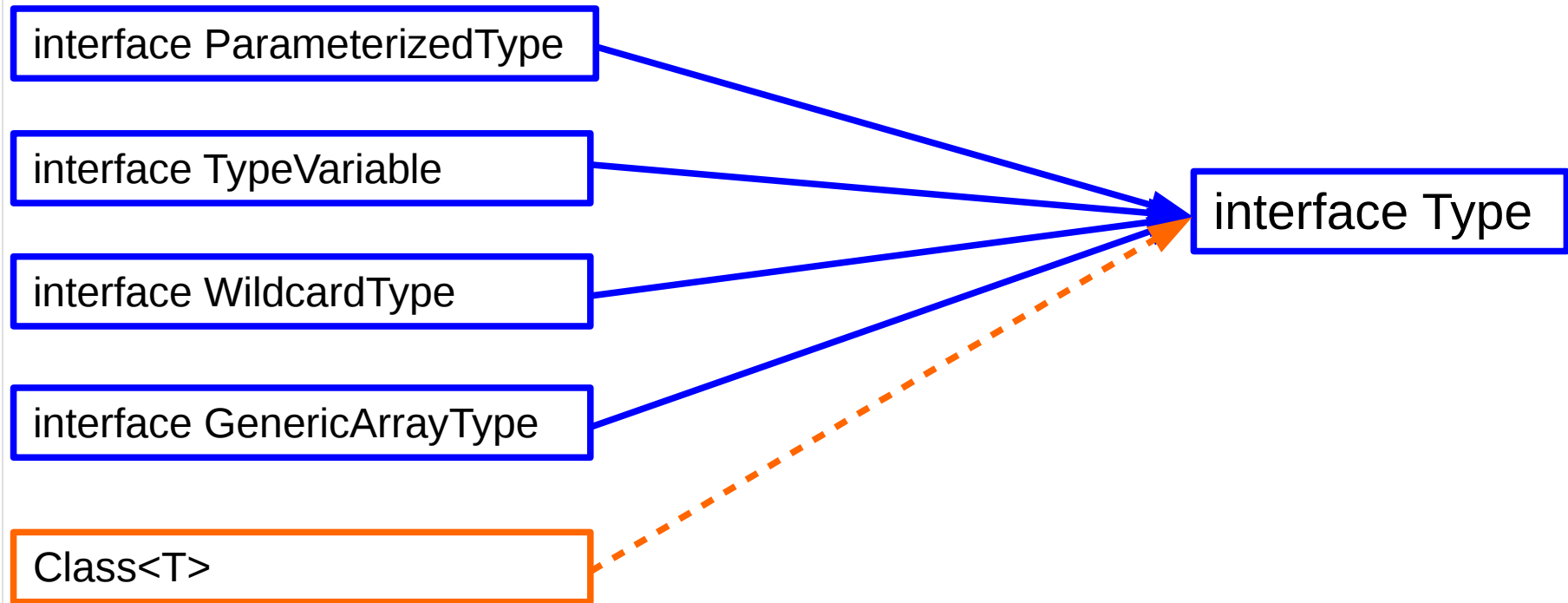
Метод	Описание
Type getRawType()	Получает базовый тип для данного параметризованного типа.
Type getOwnerType()	Получает тип внешнего класса, если данный тип внутренний, или null, если этот тип внешний.
Type [] getActualTypeArguments()	Получает параметр типа, с которым был объявлен данный параметризованный тип.

## Методы классов и интерфейсов механизма рефлексии для работы с обобщенными типами

`java.lang.reflect.GenericArrayType`

Метод	Описание
Type getGenericComponentType()	Получает обобщенный тип компонента, с которым связан тип данного массива.

# Иерархия интерфейсов и классов механизма рефлексии для работы с обобщенными типами



```
package com.gmail.tsa;
```

## Пример исследования обобщенного класса

```
public class GenericClass<T> {  
    private T field;
```

Класс обобщенного типа с параметром T

```
    public GenericClass(T field) {  
        super();  
        this.field = field;  
    }
```

```
    public T getField() {  
        return field;  
    }
```

```
    public void setField(T field) {  
        this.field = field;  
    }
```

Конструктор принимает переменную обобщенного типа

```
@Override  
    public String toString() {  
        return "GenericClass [field=" + field + "];  
    }
```

```
    public <V extends Number> V plus(V a, V b) {  
        Number c;  
        c = a.doubleValue() + b.doubleValue();  
        return (V) c;  
    }
```

Обобщенный метод с ограничением по типу

```
}
```

```
package com.gmail.tsa;

import java.lang.reflect.Method;
import java.lang.reflect.TypeVariable;
```

```
public class Main {
```

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

```
        GenericClass<String> someClass = new GenericClass<String>("Hello world");
```

```
        Class<?> genericClass = someClass.getClass();
```

```
        TypeVariable<?>[] typeArray = genericClass.getTypeParameters();
```

```
        for (TypeVariable<?> typeVariable : typeArray) {
            System.out.println(typeVariable);
        }
```

```
    try {
```

```
        Method classMethod = genericClass.getMethod("plus", Number.class,
        Number.class);
```

```
        TypeVariable<?>[] methodTypeArray = classMethod.getTypeParameters();
```

```
        for (TypeVariable<?> typeVariable : methodTypeArray) {
            System.out.println(typeVariable + " - " + typeVariable.getBounds()[0]);
        }
```

```
    } catch (NoSuchMethodException | SecurityException e) {
        e.printStackTrace();
    }
```

```
}
```

```
}
```

## Исследование обобщенного класса средствами рефлексии

Получение параметров обобщенного типа

Получение ограничения на параметр

Получение параметров обобщенного метода

## Краткие итоги урока

**Рефлексия** — инструмент для динамического исследования классов. С ее помощью можно узнать данные о свойствах, методах, конструкторах и реализуемых интерфейсах класса. Также есть возможность получить значение свойств объекта или установить их (в частности можно менять значения `private` свойств).

С помощью рефлексии можно вызвать любой метод класса, включая конструкторы. Это дает возможность динамически создавать новые объекты произвольных классов.

Начало использования рефлексии - это получение переменной типа `Class`. Объекты этого типа обладают методами доступа к свойствам, методам, конструкторам и т. д. исследуемого класса.

Рефлексия используется в случае написания приложений, которые конфигурируются с помощью внешних файлов.

## Дополнительная литература по теме данного урока

- Брюс Эккель Философия Java. 4-е издание. стр. 352-394
- Герберт Шилд. Java. Полное руководство 8-е издание. стр. 306 — 316
- Кей Хорстман, Гари Корнел Библиотека профессионала. Java. Том 1 — 9 издание. стр. 246 - 267

# Annotations

## Дополнительный материал к лекции «Полиморфизм»

Аннотация — в языке Java специальная форма синтетических метаданных, которая может быть добавлена в исходный код.

Аннотированы могут быть пакеты, классы, методы, переменные и параметры.



# Основная область использования аннотаций

## 1. Контроль за качеством кода

**@Override** - аннотация-маркер, которая может применяться только к методам. Метод, аннотированный как **@Override**, должен переопределять метод супер класса.

**@Deprecated** - указывает, что объявление устарело и должно быть заменено более новой формой.

**@SuppressWarnings** - эта аннотация указывает, что одно или более предупреждений, которые могут быть выданы компилятором, следует подавить.

**@SafeVarargs** - аннотация-маркер, применяется к методам и конструкторам. Она указывает, что никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Применяется только к методам и конструкторам с переменным количеством аргументов, которые объявлены как **static** или **final**.

## 2. Как метка для различных инструментов Java

**@Entity** — сущность для фреймворка Hibernate.

**@Controller** — указывает что класс является контролером для Spring MVC.

## 3. Как инструмент при создании собственных аннотаций

**@Retention** - эта аннотация предназначена для применения только в качестве аннотации к другим аннотациям. Определяет политику удержания.

**@Documented** - это маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована.

# Синтаксис объявления и использования аннотаций

Аннотация начинается с символа @

Аннотировать объект - это указать аннотацию выше него. Аннотация действует только на один объект ниже ее. Остальных нижележащих объектов она не касается

## Пример аннотации и аннотированного объекта

@FunctionalInterface

Аннотация впервые появившаяся в JDK 1.8 указывает на то, что аннотированный объект, является функциональным интерфейсом

```
interface Func {  
    public int get(int n);  
}
```

Аннотированный объект. Аннотация описана выше него.

Аннотации бывают встроенного и пользовательского типа. Разница между ними невелика. Встроенные - это просто заранее описанные аннотации в базовой библиотеке Java.

## Создание пользовательских аннотаций

Для создания пользовательской аннотации нужно:

- 1) Импортировать пакет `java.lang.annotation.*`.
- 2) Указать параметры связанные с аннотацией.
- 3) Указать, что будет целью данной аннотации.
- 4) Указать политику удержания данной аннотации.
- 5) Дать имя аннотации, используя механизм интерфейсов.
- 6) Указать параметры аннотации.

## Указание параметров, связанных с пользовательской аннотацией

Выполняется с помощью таких аннотаций:

**@Documented** — указывает на то, что аннотация обязательно должна быть документирована.

**@Inherited** — указывает на то, что данная аннотация будет унаследована подклассом текущего класса.

**@Repeatable** - создает повторяющуюся аннотацию (начиная с Java 8).

## Указание цели пользовательской аннотации

Выполняется с помощью встроенной аннотации **@Target**.

Эта аннотация в качестве параметра принимает одну из определенных ранее констант в перечислении `ElementType`. Параметр аннотации указывается в круглых скобках после нее.

Параметр	Возможная цель
<code>ElementType.ANNOTATION_TYPE</code>	Аннотация
<code>ElementType.CONSTRUCTOR</code>	Конструктор класса
<code>ElementType.FIELD</code>	Свойство класса или элемент перечисления
<code>ElementType.LOCAL_VARIABLE</code>	Локально определенная переменная
<code>ElementType.METHOD</code>	Метод класса
<code>ElementType.PACKAGE</code>	Пакет
<code>ElementType.PARAMETER</code>	Параметр метода
<code>ElementType.TYPE</code>	Класс, интерфейс, аннотация, перечисление
<code>ElementType.TYPE_PARAMETER</code>	Параметр обобщенного типа (начиная с Java 8).
<code>ElementType.TYPE_USE</code>	Тип данных (начиная с Java 8).

Например `@Target(ElementType.METHOD)` — аннотация, целью которой будет метод

**Если аннотация может иметь несколько типов целей, то они перечисляются через запятую** - `@Target(ElementType.METHOD, ElementType.FIELD)`

## Указание политики удержания пользовательской аннотации

Политика удержания определяет в какой точке аннотация отбрасывается. Задается с помощью встроенной аннотации **@Retention**. В качестве параметра этой аннотации указывается одна из констант перечисления `RetentionPolicy`.

Параметр	Политика удержания
<code>RetentionPolicy.SOURCE</code>	Содержатся только в исходном коде. При компиляции отбрасываются.
<code>RetentionPolicy.CLASS</code>	Сохраняются в файлах <code>class</code> , однако не доступны JVM во время выполнения.
<code>RetentionPolicy.RUNTIME</code>	Сохраняются в файлах <code>class</code> , доступны JVM во время выполнения.

Например, `@Retention(RetentionPolicy.RUNTIME)` укажет на то, что аннотация будет доступна во время выполнения.

## Описание пользовательской аннотации

Описание аннотации начинается с использования конструкции вида `@interface`.

После этого идет имя аннотации. Следующим элементом будет тело аннотации, заключенное в фигурные скобки.

### Пример описание пользовательской аннотации

Имя аннотации



```
public @interface MyAnnotation {  
}
```

## Описание параметров пользовательской аннотации

Описание параметров аннотации похоже на описание метода, который не имеет параметра и возвращает свойство указанного типа.

Например, если нужно задать два параметра пользовательской аннотации. первый имеет тип **String**, второй тип **long**, то аннотация могла бы выглядеть так:

```
public @interface MyAnnotation {  
    String paramOne(); ← Первый параметр  
    long paramTwo(); ← Второй параметр  
}
```

**!** Можно задать значение параметра по умолчанию, используя оператор **default**, за которым следует значение, которое примет параметр аннотации по умолчанию. Например, для 1-го параметра, приведенного выше, описание могло выглядеть как

```
String paramOne() default "Hello world";
```



## Ограничения на параметры пользовательской аннотации

Параметры могут иметь только следующие типы:

- Прimitives;
- String;
- Class или «any parameterized invocation of Class»;
- enum;
- annotation;
- массив элементов (одномерный) любого из вышеперечисленных типов;

## Общий вид пользовательской аннотации

```
package com.gmail.tsa;  
import java.lang.annotation.Documented;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;
```

@Documented

Доступна на этапе выполнения

@Retention(RetentionPolicy.*RUNTIME*)

@Target(ElementType.*METHOD*)

Цель аннотации — метод

**public @interface** MyAnnotation {

Имя аннотации

String paramOne() **default** "Hello world";

**long** paramTwo();

}

Имеет два параметра, первый имеет значение по умолчанию

## Пример аннотирования пользовательской аннотацией

```
package com.gmail.tsa;
```

```
public class SomeClass {
```

Задание параметров аннотации

```
@MyAnnotation(paramOne = "Go-Go", paramTwo = 15)
```

Аннотирование

```
public void printSomeText(String text, long count) {  
    for (int i = 0; i < count; i++) {  
        System.out.println(text);  
    }  
}
```

Аннотируемый метод.

В этом примере мы проаннотируем (созданной ранее аннотацией @MyAnnotation) метод произвольного пользовательского класса. И зададим параметры аннотации.

## Использование пользовательских аннотаций

Основной метод работы с пользовательскими аннотациями — это использование рефлексии. Т.е., сначала аннотируются нужные члены класса (или сам класс), а потом, с помощью рефлексии, обнаруживаются наличие аннотации. При необходимости ее можно извлечь для получения ее параметров и дальнейшего использования.

Метод	Описание
<code>&lt;T extends Annotation&gt; T getAnnotation(Class&lt;T&gt; тип аннотации)</code>	Возвращает ссылку на аннотацию (используя эту ссылку можно получить параметры аннотации).
<code>boolean isAnnotationPresent(Class&lt;? extends Annotation&gt; annotationClass)</code>	Вернет true, если объект аннотирован указанной аннотацией, и false, если нет.
<code>Annotation[] getAnnotations()</code>	Вернет все аннотации, ассоциированные с данным объектом в том числе и унаследованные.
<code>Annotation[] getDeclaredAnnotations()</code>	Вернет все аннотации (ассоциированные напрямую с данным объектом).

# Пример получения аннотации средствами рефлексии

```
package com.gmail.tsa;

import java.lang.reflect.Method;

public class Main {

    public static void main(String[] args) {

        Class<?> someClass = SomeClass.class;
        try {
            Method method = someClass.getDeclaredMethod("printSomeText",
String.class, long.class);

            if (method.isAnnotationPresent(MyAnnotation.class)) {

                MyAnnotation myAnnotation =
method.getAnnotation(MyAnnotation.class);

                System.out.println(myAnnotation.paramOne());
                System.out.println(myAnnotation.paramTwo());

            }

        } catch (NoSuchMethodException e) {

        }

    }

}
```

Проверка наличия аннотации

if (method.isAnnotationPresent(MyAnnotation.class)) {

MyAnnotation myAnnotation =  
method.getAnnotation(MyAnnotation.class);

Получение

System.out.println(myAnnotation.paramOne());  
System.out.println(myAnnotation.paramTwo());

Извлечение параметра аннотации

# Использование аннотаций в прикладных целях

Постановка задачи: вызвать проаннотированный метод и передать ему параметры аннотации.

```
package com.gmail.tsa;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Main {

    public static void main(String[] args) {

        SomeClass a = new SomeClass();

        Class<?> someClass = a.getClass();
        try {
            Method method = someClass.getDeclaredMethod("printSomeText", String.class,
                long.class);

            MyAnnotation myAnnotation = method.getAnnotation(MyAnnotation.class);

            method.invoke(a, myAnnotation.paramOne(), myAnnotation.paramTwo());

        } catch (NoSuchMethodException | IllegalAccessException | IllegalArgumentException
            | InvocationTargetException e) {

        }

    }

}
```

Получение аннотации

Вызов метода с передачей ему параметров аннотации

## Специализированные виды аннотаций

Повторяющиеся — один элемент может быть проаннотирован несколькими одинаковыми аннотациями. Доступно только начиная с JDK 1.8

Для создания повторяющейся аннотации ее следует проаннотировать аннотацией **@Repeatable**. Ее значением указывается тип контейнера. Такой контейнер указывается в виде аннотации, для которой поле value является массивом типа повторяющейся аннотации. Следовательно, чтобы сделать аннотацию повторяющейся, прежде нужно создать контейнерную аннотацию, а затем указать ее тип в качестве аргумента аннотации **@Repeatable**.

Для доступа к повторяющимся аннотациями с помощью такого метода, как, например, `getAnnotation()`, следует воспользоваться контейнерной, а не самой повторяющейся аннотацией.

## Пример повторяющихся аннотаций

```
package com.gmail.tsa;
```

```
import java.lang.annotation.ElementType;  
import java.lang.annotation.Repeatable;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;
```

```
@Repeatable(MyAnnotationConteyner.class)
```

Аннотация повторяющаяся

```
@Retention(RetentionPolicy.RUNTIME)
```

Ссылка на контейнер аннотации

```
@Target(ElementType.METHOD)
```

```
public @interface MyAnnotation {
```

```
    String paramOne() default "Hello world";
```

```
    long paramTwo();
```

```
}
```

Аннотация, указывающая на  
контейнер повторяющейся аннотации

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.METHOD)
```

```
@interface MyAnnotationConteyner {
```

```
    MyAnnotation[] value();
```

Контейнером выступает массив

```
}
```



## Пример повторяющихся аннотаций

```
package com.gmail.tsa;
```

```
public class SomeClass {
```

```
    @MyAnnotation(paramOne = "Go-Go", paramTwo = 5)  
    @MyAnnotation(paramOne = "Hello", paramTwo = 5)
```

```
    public void printSomeText(String text, long count) {  
        for (int i = 0; i < count; i++) {  
            System.out.println(text);  
        }  
    }  
}
```

Двойное аннотирование метода



## Пример повторяющихся аннотаций

```
package com.gmail.tsa;  
  
import java.lang.reflect.InvocationTargetException;  
import java.lang.reflect.Method;
```

```
public class Main {
```

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

```
        SomeClass a = new SomeClass();
```

```
        Class<?> someClass = a.getClass();
```

```
        try {  
            Method method = someClass.getDeclaredMethod("printSomeText", String.class,  
                long.class);
```

```
            MyAnnotationConteyner an = method.getAnnotation(MyAnnotationConteyner.class);
```

```
            method.invoke(a, an.value()[0].paramOne(), an.value()[0].paramTwo());  
            method.invoke(a, an.value()[1].paramOne(), an.value()[1].paramTwo());
```

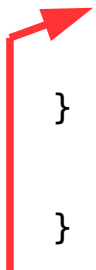
```
        } catch (NoSuchMethodException | IllegalAccessException | IllegalArgumentException  
            | InvocationTargetException e) {
```

```
        }
```

```
    }
```

```
} Получение массива повторных аннотаций и выполнение методов с  
их параметрами
```

Получение аннотации контейнера



## Итоги урока

**Аннотация** — вид метаданных в Java. Все аннотации реализуют интерфейс **Annotation**.

У аннотаций есть такой параметр как цель аннотации (может быть несколько):

- ElementType.ANNOTATION\_TYPE
- ElementType.CONSTRUCTOR
- ElementType.FIELD
- ElementType.LOCAL\_VARIABLE
- ElementType.METHOD
- ElementType.PACKAGE
- ElementType.PARAMETER
- ElementType.TYPE
- ElementType.TYPE\_PARAMETER
- ElementType.TYPE\_USE

Политика удержания (одна):

- RetentionPolicy.SOURCE
- RetentionPolicy.CLASS
- RetentionPolicy.RUNTIME

Так же аннотации могут быть снабжены параметрами пользователя и части из них можно дать значение по умолчанию.

Аннотации можно исследовать методами рефлексии. При этом можно как обнаружить аннотированные члены класса, так и получить параметры этой аннотации.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 305 - 316.

# Java OOP

## (Интерфейсы)

**Интерфейс** — синтаксическая конструкция в коде программы, используемая для специфицирования услуг, предоставляемых классом или компонентом.

**Интерфейс** определяет границу взаимодействия между классами или компонентами, специфицируя определенную абстракцию, которую осуществляет реализующая сторона.

Для объявления интерфейса используется  
ключевое слово **interface**

Использование интерфейсов позволяет абстрагировать  
описательную часть класса от его реализации.

Синтаксис объявления интерфейса в Java

```
Модификатор доступа interface имя {  
  
    Тип_возвращаемой_переменной имя_метода (список параметров);  
  
    Тип_переменной имя_переменной = значение;  
    .....  
}
```

**!** Методы, описанные в интерфейсах, не имеют тела

# Некоторые аспекты при объявлении интерфейсов

Переменные, объявленные внутри интерфейса, неявно объявляются как `final` и `static`. Они должны быть инициализированы. Все методы неявно объявлены как `public`.

В отличие от абстрактных классов в интерфейсе не может быть ни одного метода с реализацией (до Java 8).

# Реализация интерфейсов

Для того чтобы класс реализовал один или несколько интерфейсов, используется ключевое слово **implements**.

Общий вид описания класса, реализующего интерфейс:  
Модификатор\_доступа class Имя\_класса **implements** Имя\_интерфейса

Класс может реализовать более одного интерфейса.  
Для этого требуется перечислить реализуемые интерфейсы  
через запятую.

Если класс реализует интерфейс не полностью, то он может  
быть только абстрактным.



# Примеры объявления интерфейсов

```
package com.gmail.tsa;  
  
public interface Arithmetic {  
  
    double plus(double a, double b);  
    double minus(double a, double b);  
    double div(double a, double b);  
    double mul(double a, double b);  
}
```



Методы, которые нужно  
будет реализовать



```
package com.gmail.tsa;  
  
public interface Info {  
    void printinfo();  
}
```

## Пример класса реализующего один интерфейс

```
package com.gmail.tsa;

public class Num implements Arithmetic {

    public double plus (double a, double b){
        return a+b;
    }
    public double minus (double a, double b){
        return a-b;
    }
    public double mul(double a, double b){
        return a*b;
    }
    public double div(double a, double b){
        if(b!=0) return a/b;
        else
            return 0;
    }
}
```

Реализация методов  
определенных  
в интерфейсе

# Класс реализующий два интерфейса

`package` com.gmail.tsa; Теперь реализовать нужно два интерфейса

`public class` Num `implements` Arithmetic, Info {

`public double` plus (`double` a, `double` b){  
    `return` a+b;

}

`public double` minus (`double` a, `double` b){  
    `return` a-b;

}

`public double` mul(`double` a, `double` b){  
    `return` a\*b;

}

`public double` div(`double` a, `double` b){  
    `if`(b!=0) `return` a/b;  
    `else`  
        `return` 0;

}

`public void` printinfo(){  
    System.out.println("HELLO");

}

}

Методы интерфейса  
Arithmetic

Метод интерфейса Info

## Переменная интерфейса может ссылаться на объект реализующего класса

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        Num a=new Num();  
        System.out.println(a.plus(3.4, 5.6));  
  
        Arithmetic b=a; ←  
        Info c=a;  
  
        System.out.println(b.plus(56, 34));  
  
        c.printinfo();  
    }  
}
```

Переменные интерфейса ссылаются на объекты, реализующего их, класса

Использование методов с помощью переменной интерфейса

# Вложенные интерфейсы

Интерфейс может быть объявлен внутри какого-нибудь класса. Такой интерфейс называется вложенным.

```
package com.gmail.tsa;  
  
public class TopClass {  
    int x;  
  
    public interface IsNum{  
        boolean isNumber(String a);  
    }  
}
```



Вложенный интерфейс

# Пример реализации вложенного интерфейса

Реализация вложенного интерфейса



```
package com.gmail.tsa;

public class PrB implements TopClass.IsNum{
    public boolean isNumber (String t){
        boolean flag=false;
        try{
            Double.valueOf(t);
            flag=true;
        }
        catch(Exception e){

        }
        return flag;
    }
}
```

Интерфейсы также могут наследовать друг друга. В таком случае, чтобы реализовать такой интерфейс, нужно реализовать все методы super интерфейса и подинтерфейса.

```
package com.gmail.tsa;
```

Наследование интерфейса интерфейсом



```
public interface PodInt extends Info{
```

```
    void print2t();
```

```
}
```

## Новое в интерфейсах начиная с Java 8

- 1) Теперь в интерфейсах можно объявлять `static` методы.
- 2) Появилась возможность объявления `default` методов. `default` метод — метод, уже реализованный в теле интерфейса. Если в классе реализующем интерфейс, он не будет переопределен, то он будет использоваться по умолчанию.
- 3) Появилась концепция «функционального» интерфейса и возможность использования лямбда функций для их реализаций. Функциональный интерфейс - это интерфейс, обладающий только одним абстрактным методом.



## Пример описания интерфейсов Java 8

```
package com.gmail.tsa;
```

```
public interface InterfaseNew {
```

```
    static void printHello() {  
        System.out.println("Hello");  
    }
```



Объявление static метода

```
    default int sum(int a, int b) {  
        return a + b;  
    }
```



Объявление default метода

```
    int mul(int a, int b);  
}
```



Объявление обычного метода без реализации

## Пример класса реализующего интерфейс в Java 8

```
package com.gmail.tsa;
```

```
public class A implements InterfaseNew{
```

```
    @Override
```

```
    public int mul(int a, int b) { ← Переопределение обычного метода
```

```
        return a*b;
```

```
    }
```

```
}
```

Теперь, если вы не реализуете default методы, то будет использоваться их реализация по умолчанию.

## Работа с классом реализующего интерфейс в Java 8

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
InterfaseNew.printHello();
```



Использование static метода


```
InterfaseNew a=new A();
```



Присвоение переменной интерфейса  
Класса его реализующего

```
System.out.println(a.sum(3, 4));
```

```
System.out.println(a.mul(3, 4));
```



Использование default метода

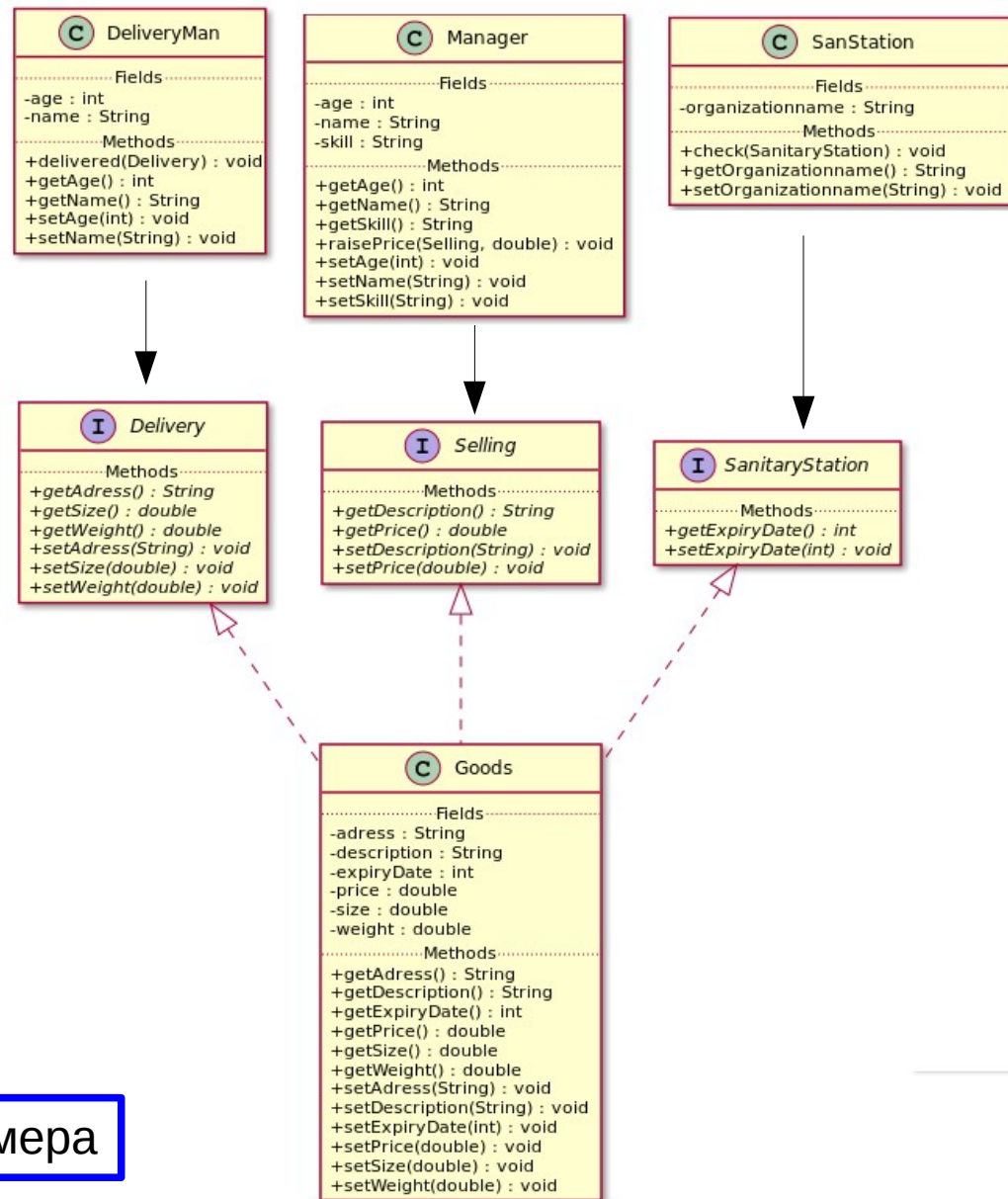
```
}
```

```
}
```



Использование переопределенного метода

# Пример использования интерфейсов для разделения доступа



Структура используемого примера

```
package com.gmail.tsa;
```

## Реализуемые интерфейсы

```
public class Goods implements Selling, Delivery, SanitaryStation {
```

```
    private double price;  
    private double size;  
    private double weight;  
    private String description;  
    private String address;  
    private int expiryDate;
```

Класс описывающий товар и реализующий интерфейсы продавца, курьера, и проверяющего сан. службы.

```
    public Goods(double price, double size, double weight, String description, String address, int expiryDate) {  
        this.price = price;  
        this.size = size;  
        this.weight = weight;  
        this.description = description;  
        this.address = address;  
        this.expiryDate = expiryDate;  
    }
```

Методы, реализующие заданные интерфейсы



```
    public double getPrice() { return price; }  
    public void setPrice(double price) { this.price = price; }  
    public double getSize() { return size; }  
    public void setSize(double size) { this.size = size; }  
    public double getWeight() { return weight; }  
    public void setWeight(double weight) { this.weight = weight; }  
    public String getDescription() { return description; }  
    public void setDescription(String description) { this.description = description; }  
    public String getAddress() { return address; }  
    public void setAddress(String address) { this.address = address; }  
    public int getExpiryDate() { return expiryDate; }  
    public void setExpiryDate(int expiryDate) { this.expiryDate = expiryDate; }  
}
```

# Интерфейсы для разграничения доступа к данным

## Интерфейс продавца

```
package com.gmail.tsa;

public interface Selling {

    public double getPrice();

    public void setPrice(double price);

    public String getDescription();

    public void setDescription(String
        description);
}
```

## Интерфейс курьера

```
package com.gmail.tsa;

public interface Delivery {

    public double getSize();

    public void setSize(double size);

    public double getWeight();

    public void setWeight(double
        weight);

    public String getAdress();

    public void setAdress(String
        adress);
}
```

## Интерфейс проверяющего

```
package com.gmail.tsa;

public interface SanitaryStation {

    public int getExpiryDate();

    public void setExpiryDate(int expiryDate);
}
```

## Класс «DeliveryMan»

```
package com.gmail.tsa;
public class DeliveryMan {
    private String name;
    private int age;

    public DeliveryMan(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void delivered(Delivery d) {
        System.out.println("Goods delivered to the address: " +
            d.getAdress());
    }
}
```

Интерфейс «Delivery» выступает параметром метода

## Класс «Manager»

```
package com.gmail.tsa;

public class Manager {
    private String name;
    private int age;
    private String skill;

    public Manager(String name, int age, String skill) {
        super();
        this.name = name;
        this.age = age;
        this.skill = skill;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSkill() {
        return skill;
    }

    public void setSkill(String skill) {
        this.skill = skill;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void raisePrice(Selling s, double a){
        s.setPrice(s.getPrice()+a);
    }
}
```

Интерфейс «Selling» выступает параметром метода



Обратите внимание, что все классы используют в качестве параметров объекты соответствующих интерфейсов. Использование интерфейсов позволяет разграничить доступ к методам класса «Good».



## Класс «SanStation»

```
package com.gmail.tsa;
```

```
import java.util.Calendar;
```

```
public class SanStation {  
    private String organizationname;  
  
    public SanStation(String organizationname) {  
        super();  
        this.organizationname = organizationname;  
    }
```

```
    public String getOrganizationname() {  
        return organizationname;  
    }
```

```
    public void setOrganizationname(String organizationname) {  
        this.organizationname = organizationname;  
    }
```

Интерфейс «SanitaryStation» выступает параметром метода



```
    public void check(SanitaryStation ss) {  
        if (ss.getExpiryDate() < Calendar.getInstance().get(Calendar.YEAR)) {  
            System.out.println("Expiry date of your product has expired");  
        } else {  
            System.out.println("Your goods pass");  
        }  
    }  
}
```

## Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

Создаем объект класса Goods

```
Goods corn = new Goods(20, 20, 300, "fresh corn", "Kiev", 2015);
```

```
Manager m = new Manager("Danil", 23, "Novice");
```

```
m.raisePrice(corn, 15);
```

```
System.out.println(corn.getPrice());
```

```
DeliveryMan dm = new DeliveryMan("Igor", 35);
```

```
dm.delivered(corn);
```

```
SanStation ss1 = new SanStation("KIEV SES");
```

```
ss1.check(corn);  
}
```

И хотя параметром метода является объект класса Goods, он приведет к соответствующему интерфейсу.

```
}
```

## Пример реализации встроенных интерфейсов

Реализация интерфейса **Comparable** (сравниваются текущий объект и передаваемый как параметр).

Для чего нужна реализация этого интерфейса?

Для использования алгоритма сортировки массивов объектов стандартными способами или для возможности сравнить объекты.

# Пример класса реализующего интерфейс Comparable

```
package com.gmail.tsa;
```

```
public class UsbDrive implements Comparable {
```

```
    private int size;
```

```
    private String brand;
```

```
    private int type;
```

```
    public UsbDrive(int size, String brand, int type) {
```

```
        super();
```

```
        this.size = size;
```

```
        this.brand = brand;
```

```
        this.type = type;
```

```
    }
```

```
    public int getSize() {
```

```
        return size;
```

```
    }
```

```
    public void setSize(int size) {
```

```
        this.size = size;
```

```
    }
```

```
    public String getBrand() {
```

```
        return brand;
```

```
    }
```

```
    public void setBrand(String brand) {
```

```
        this.brand = brand;
```

```
    }
```

```
    public int getType() {
```

```
        return type;
```

```
    }
```

```
    public void setType(int type) {
```

```
        this.type = type;
```

```
    }
```

Класс реализует интерфейс Comparable,  
можно сортировать стандартными методами

Здесь должны быть методы реализующие эти интерфейсы  
(они вынесены на следующий слайд).

## Методы, реализующие указанные интерфейсы (смотри предыдущий слайд)

```
@Override  
public int compareTo(Object o)
```

Для метода сравнения параметрами  
выступают объекты типа Object

```
    if (o == null) {  
        return -1;  
    }
```

```
    UsbDrive anotherdrive = (UsbDrive) o;
```

Сравнение будет по  
имени производителя

```
    return this.brand.compareToIgnoreCase(anotherdrive.getBrand());  
}
```

```
@Override  
public String toString() {  
    return "UsbDrive [size=" + size + ", brand=" + brand + ",  
    type=" + type + "];"  
}
```

## Пример использования стандартных методов для массивов объектов

```
package com.gmail.tsa;
```

```
import java.util.Arrays;
```

```
public class Main {
```

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

Создание четырех UsbDrive

```
        UsbDrive fl1 = new UsbDrive(16, "Transdent", 2);  
        UsbDrive fl2 = new UsbDrive(4, "Kingston", 3);  
        UsbDrive fl3 = new UsbDrive(32, "Samsung", 2);  
        UsbDrive fl4 = new UsbDrive(1, "Silicon power", 2);
```

```
        UsbDrive[] usbmass = { fl1, fl2, fl3, fl4 };
```

```
        Arrays.sort(usbmass);
```

Сортировка массивов UsbDrive

```
        for (UsbDrive usbDrive : usbmass) {  
            System.out.println(usbDrive);  
        }
```

Вывод массива UsbDrive на экран

```
    }
```

## Реализация интерфейса Cloneable и создание копии объекта

Для создания копии объекта (а не еще одной ссылки на объект) нужно реализовать интерфейс супер класса Object - **Cloneable**, для этого необходимо переопределить метод

- protected Object clone()

Правила, по которым работает метод clone()

`x.clone() != x` -> true (т.к. разные объекты)

`x.clone().getClass() == x.getClass()` -> true (одинаковый класс)

`x.clone().equals(x)` -> true (реже false)

## Пример класса реализующего интерфейс Cloneable

```
package com.gmail.tsa;
```

```
public class Rectangle implements Cloneable {
```


```
    double width;  
    double length;
```

```
    public Rectangle(double width, double length) {  
        super();  
        this.width = width;  
        this.length = length;  
    }
```

```
@Override
```

```
public Rectangle clone() {  
    try {  
        return (Rectangle) super.clone();  
    } catch (CloneNotSupportedException e) {  
        return null;  
    }  
}
```

Использование метода суперкласса Object



Реализация метода  
clone()



Исключени,е которое может вызвать метод



## Пример создания копии объекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Rectangle a = new Rectangle(4, 5);
```

```
        Rectangle b = new Rectangle(4, 5);
```

```
        Rectangle c = a.clone();
```

← Копирование объекта a в объект c

```
        a.width = 3;
```

← Изменение свойства объекта a

```
        System.out.println(a.width);  
        System.out.println(c.width);
```

← Вывод на экран свойств объектов

```
    }  
}
```

Внимание! Стандартный метод clone() реализует только «поверхностную копию», т. е., все примитивные типы копируются по значению, а для ссылочных переменных копируются только ссылки.

## Краткие итоги урока

**Интерфейс** — синтаксическая конструкция в коде программы, используемая для специфицирования услуг, предоставляемых классом или компонентом. Т.е. по сути использовать интерфейс нужно в случае необходимости указания того, что умеет делать сущность или что можно делать с ней.

Интерфейсы описываются с помощью ключевого слова **interface**, а классы которые удовлетворяют спецификации интерфейса (реализуют все его абстрактные методы) описываются с помощью ключевого слова **implements**.

Ранг интерфейса выше ранга класса его реализующего. Следовательно, переменную интерфейса всегда можно присвоить объекту класса, который данный интерфейс реализует. В таком случае переменная интерфейса имеет доступ только к тем методам, которые объявлены в интерфейсе.

Начиная с JDK 1.8 появилось несколько **нововведений** в интерфейсах:

- 1) Наличие статических методов;
- 2) Реализация по умолчанию;
- 3) Функциональные интерфейсы;

Реализация встроенных интерфейсов необходима для более тесной интеграции ваших классов со стандартными методами JDK. Например, это сортировка и клонирование объектов вашего типа.

## Дополнительная литература по теме данного урока.

- Брюс Эккель Философия Java. 4-е издание. стр. 221-244
- Герберт Шилд. Java. Полное руководство 8-е издание. стр. 244 - 261
- Кей Хорстман, Гари Корнел Библиотека профессионала. Java. Том 1 - 9 издание. стр. 271 - 286

## Домашнее задание

1. Усовершенствуйте класс Group, добавив возможность интерактивного добавления объекта.
2. Реализуйте возможность сортировки списка студентов по фамилии.
3. Реализуйте возможность сортировки по параметру (Фамилия, успеваемость и т. д.).
4. Реализуйте интерфейс Военком, который вернет из группы массив студентов - юношей, которым больше 18 лет.
5. Протестируйте его работу.

# Функциональные интерфейсы и лямбда функции. Дополнительный материал к лекции «Интерфейсы»

Функциональным интерфейсом называют интерфейс с единственным абстрактным методом, который и является типом лямбда-выражения. Т.е, функциональный интерфейс обычно описывает одно действие.

Составил: Цымбалюк А.Н.

## Пример функционального интерфейса

### Обычный

```
@FunctionalInterface
interface Func {
    public int get(int n);
}
```

### Обобщенный

```
@FunctionalInterface
interface SomeFunc<T>{
    public T get(T t);
}
```

@FunctionalInterface – Новая аннотация, появившаяся в JDK 1.8. Интерфейс, который будет объявлен после этой аннотации, обязан быть функциональным. Т.е., иметь только один нереализованный метод.

## Нововведения в JDK 1.8, связанные с появлением функциональных интерфейсов

Нововведением в JDK 1.8 является то, что теперь функциональный интерфейс может напрямую ссылаться на метод объекта или класса, который его реализует, причем не обязательно совпадение названий методов.

Для этого используется оператор ::

- Если нужно сослаться на статический метод то, Имя класса :: метод
- Если на не статический метод то, Имя объекта :: метод
- Если нужно сослаться на конструктор то, Имя класса :: new

```
package com.gmail.tsa;
```

Пример использования функционального интерфейса

```
@FunctionalInterface  
interface Func {  
    public int get(int n);  
}
```

Объявление функционального интерфейса

```
public class Main {
```

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

```
        Func func;
```

Объявление переменной функционального интерфейса

```
        func = A::getStaticMethod;  
        System.out.println(func.get(8));
```

Ссылка на статический метод

```
        A a = new A();
```

Создание переменной класса реализующего интерфейс

```
        func = a::getMethod;  
        System.out.println(func.get(8));
```

Ссылка на обычный метод

```
    }  
}
```

```
class A {  
    public static int getStaticMethod(int n) {  
        return 3 * n;  
    }  
    public int getMethod(int n) {  
        return 5 * n;  
    }  
}
```

Статический


Обычный

Методы, реализующие функциональный интерфейс




Примечание: как видно, достаточно чтобы при реализации совпадал только тип возвращаемого значения и список параметров. Совпадение имен метода в функциональном интерфейсе и метода, на который можно сослаться, не обязательно.

```
@FunctionalInterface  
interface Func {  
    public int get(int n);  
}
```



Имя метода в функциональном интерфейсе

Имя метода, на который может сослаться функциональный интерфейс



```
public static int getStaticMethod(int n) {  
    return 3 * n;  
}
```

В этом случае совпадать должно только то, что метод возвращает int, и что у него один параметр типа int

Крупным нововведением в JDK 1.8 является появление лямбда-функций. Лямбда-функция - это анонимная реализация функционального интерфейса, которая является объектом. Т. е. может быть передана по ссылке, быть параметром метода и т.д.

Для введения дополнительного функционала в JDK 1.8 введен новый оператор -> который и идентифицирует лямбда-функцию.

- Лямбда-функция является блоком кода с параметрами.
- Используйте лямбда-функции, когда хотите выполнить блок кода в более поздний момент времени.
- Лямбда-функция может быть преобразована в функциональные интерфейсы.
- Лямбда-функция имеет доступ к `final` переменным из охватывающей области видимости.
- Ссылки на метод и конструктор ссылаются на методы или конструкторы без их вызова.
- Теперь вы можете добавить методы по умолчанию и статические методы к интерфейсам, которые обеспечивают конкретные реализации.
- Вы должны разрешать любые конфликты между методами по умолчанию из нескольких интерфейсов.

## Пример использования лямбда-функции

```
package com.gmail.tsa;
```

Объявление функционального интерфейса

```
@FunctionalInterface  
interface Func {  
    public int get(int n);  
}
```

```
public class Main {
```

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

```
        Func func;
```

Реализация с помощью лямбда-функции

```
        func = n -> n * n;
```

```
        System.out.println(func.get(5));
```

```
    }  
}
```

## О синтаксисе лямбда-функции

Оператор, обозначающий отображение



```
func = n -> n * n;
```

Параметры метода  
(должны совпадать и тип,  
и количество)

То, что метод должен возвращать тип,  
также должен совпадать

О типах параметров: Обычно тип параметров выводится автоматически и не нуждается в явном указании. Однако, в случае двух двузначности, тип нужно указать явно.

Явное указание типа параметра

```
func = (int n) -> n * n;
```

## Более сложный синтаксис лямбда-функции

В случае, если лямбда функция состоит из большой последовательности операторов, используется другой синтаксис.

```
func = n -> {  
    int fact = 1;  
    for (int i = 1; i <= n; i++) {  
        fact *= i;  
    }  
    return fact;  
};
```

Тело метода заключается в {}

В таком случае, оператор return обязателен

; - обязательны после закрытия тела реализации

## Ограничения лямбда-функций

Лямбда-функции всегда являются реализациями какого-либо определенного функционального интерфейса. Лямбда-функции можно передавать только как ссылки на интерфейс; как и в случае других реализаций интерфейса, лямбда-функцию можно использовать только в качестве определенного интерфейса, для чего она и была создана.

Лямбда-функция может использовать только `final` переменные из окружающей ее области видимости.

## Правила работы с функциональными интерфейсами в Java 8

- 1) Вместо функционального интерфейса можно подставить лямбда-функцию его реализующего.
- 2) Функциональному интерфейсу можно присвоить ссылку на статический метод класса, который его реализует.
- 3) Функциональному интерфейсу можно присвоить ссылку на метод объекта, который его реализует.
- 4) Функциональному интерфейсу можно присвоить ссылку на конструктор класса (если они совместимы).

## Важные функциональные интерфейсы в Java 8

Имя интерфейса	Аргументы	Тип возвращаемого значения	Описание
Predicate<T>	T	Boolean	Соответствие T — булево значение
Consumer<T>	T	void	Операция над объектом T типа
Function<T,R>	T	R	На основе объекта T типа создается объект R типа
Supplier<T>		T	Создает объект T типа
UnaryOperator<T>	T	T	Создает объект T типа на основе объекта T типа
BinaryOperator<T>	(T,T)	T	Создает объект T на основе двух объектов T типа
Comparator<T>	(T,T)	int	Сравнение двух объектов T типа и возвращает результат сравнения в виде int

В JDK 1.8 добавлено большое количество функциональных интерфейсов



## Пример использования функционального интерфейса

Задача: есть массив целых цифр (Integer - и это важно! С примитивными типа этот интерфейс не работает) отсортировать их по возрастанию значений их модуля.

Для сортировки используется функциональный интерфейс Comparator. Переопределим его с помощью лямбда-функции.

```
package com.gmail.tsa;

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {

        Integer[] array = { -2, 1, -1, 4, -3, 5, 6, -8, 9 };

        Arrays.sort(array, (nOne, nTwo) -> Math.abs(nOne) - Math.abs(nTwo));

        System.out.println(Arrays.toString(array));

    }

}
```

Сортируемый массив

Переопределение Comparator с помощью лямбда-функции

## Итоги урока

**Функциональные интерфейсы** (интерфейс с одним нереализованным методом ) - важное нововведение в JDK 1.8, реализующее (хотя и частично) функциональную парадигму программирования.

Теперь функциональные интерфейсы могут ссылаться на методы класса, которые совпадают с типом параметров и типом возвращаемых значений (совпадение названий методом не обязательно).

**Лямбда функции** — анонимные методы, реализующие функциональный интерфейс. Для введения лямбда функции в Java ввели специальный оператор -> который означает отображение. Лямбда-функции используют автоматическое выведение типов, на основе функционального интерфейса.

Также, начиная с JDK 1.8, введено много функциональных интерфейсов, спектр использования которых полностью раскрывается при использовании Stream API.

## Дополнительная литература по теме данного урока.

- Ричард Уорбэртон Лямбда выражения в Java 8, стр 21-32
- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 437 - 465

# Java OOP

## (Потоки ввода-вывода)

**Поток** — абстракция, которая, либо порождает, либо принимает информацию. Поток связан с физическим устройством при помощи системы ввода-вывода.

# Работа с объектами файловой системы в Java

В Java за работу с объектами файловой системы (файлы и каталоги) и организацию методов по вводу и выводу информации ответственны два крупных пакета — IO, NIO.

## Основные различия

IO	NIO
Потокоориентированный	Буферориентированный
Блокирующий (синхронный) ввод/вывод	Неблокирующий (асинхронный) ввод/вывод
	Селекторы

# Основы работы с файловой системой IO

При использовании IO основным инструментом по работе с файловой системой является класс File

Расположен в пакете java.io.File

Конструктор принимает в качестве параметра абсолютный или относительный адрес объекта в виде строковой переменной

```
File f1=new File("a.txt");
```



Адрес файла

Объект, ассоциированный с файлом

Внимание! Если файл по указанному адресу существует, то объект будет связан с файлом на диске. Если же по указанному адресу файл не существовал, то объект будет связан с виртуальным файлом в ОЗУ.

## Методы класса File

Метод	Описание
String getName()	Вернет имя файла
String getParent()	Вернет имя каталога файла
String getAbsolutePath()	Вернет путь к файлу
boolean exist()	Вернет true, если файл существует
boolean renameTo(File новое имя)	Переименовывает файл
boolean delete()	Удаляет файл
boolean isDirectory()	Проверяет, является ли объект каталогом
boolean mkdir()	Создает каталог
boolean createNewFile()	Создать новый файл
boolean mkdirs()	Создает каталог, путь для которого еще не создан
String [] list()	Только для каталогов. Список остальных объектов внутри него в виде массива строк
File [] listFiles()	Возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге
long length()	Возвращает размер файла в байтах

И файлы, и каталоги в Java IO описываются с помощью объекта класса File

## Пример использования File

```
package com.gmail.tsa;
```

```
import java.io.*;
```

```
public class FileOperation {
```

```
    public static void main(String[] args) {  
        PrintWriter pw = new PrintWriter(System.out, true);
```

```
        File file1 = new File("a.txt");
```

```
        try {  
            file1.createNewFile();  
        } catch (IOException e) {  
            pw.println("ERROR!");  
        }
```

Создается пустой файл

```
        File folder = new File("A");  
        folder.mkdirs();
```

Создается пустой каталог

```
        file1.renameTo(new File("b.txt"));
```

Переименовываем файл

```
        File file2 = new File("b.txt");  
        file2.delete();
```

Удаляется файл

```
        File f1 = new File(".");  
        if (f1.isDirectory()) {  
            String[] filenames = f1.list();  
            for (String filename : filenames) {  
                pw.println(filename);  
            }  
        }
```

Получаем содержание каталога  
и выводим на экран

```
    }
```



## Использование интерфейса FilenameFilter и FileFilter

Иногда требуется ограничить список файлов которые возвращают методы `list()` и `listFiles()`. Например, включать в список не все файлы, а только файлы с указанным расширением.

Для этого можно использовать перегруженные методы:

- `list(FilenameFilter fileNameFilter);`
- `listFiles (FilenameFilter fileNameFilter);`
- `listFiles (FileFilter filefilter);`

Где:

- `FilenameFilter fileNameFilter` - объект класса, реализующий интерфейс `FilenameFilter`
- `FileFilter filefilter` - объект класса, реализующий интерфейс `FileFilter`

## Описание этих интерфейсов

В `FilenameFilter` — нужно переопределить единственный метод:

`boolean accept (File каталог, String имя_файла)`

Если указанный метод вернет `true`, значит этот файл будет включен в список.

В `FileFilter` — нужно переопределить единственный метод:

`boolean accept (File файл)`

Если указанный метод вернет `true`, значит этот файл будет включен в список.

**package** com.gmail.tsa; **Пример класса, реализующего FileFilter**

**import** java.io.File;  
**import** java.io.FileFilter;

Массив для «разрешенных расширений файлов»

**public class** MyFileFilter **implements** FileFilter {  
    **private** String[] arr;

**public** MyFileFilter(String... arr) {  
        **super**();  
        **this**.arr = arr;  
    }

Инициализация его в конструкторе

**private boolean** check(String ext) {  
        **for** (String stringExt : arr) {  
            **if** (stringExt.equals(ext)) {  
                **return true**;  
            }  
        }  
        **return false**;  
    }

Проверим, есть ли такое расширение в массиве и, если есть, то вернем true, а в противном случае false

**@Override**  
    **public boolean** accept(File pathname) {  
        **int** pointerIndex = pathname.getName().lastIndexOf(".");  
        **if** (pointerIndex == -1) {  
            **return false**;  
        }  
        String ext = pathname.getName().substring(pointerIndex + 1);  
        **return** check(ext);  
    }

Переопределение метода интерфейса. Получаем расширение у файла и сравниваем с теми, что есть в массиве

}

## Пример использования класса, реализующего FileFilter

```
package com.gmail.tsa;
```

```
import java.io.File;
```

```
public class Main {
```

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

```
        MyFileFilter mFF = new MyFileFilter("doc", "pdf");
```

```
        File folder = new File("myFolder");
```


```
        File[] fileList = folder.listFiles(mFF);
```

```
        for (File file : fileList) {  
            System.out.println(file);  
        }
```

```
    }
```

```
}
```


Создаем экземпляр класса и передаем ему список расширений.




Получаем ссылку на каталог



Использование объекта класса реализующего FileFilter в перегруженной версии метода listFiles

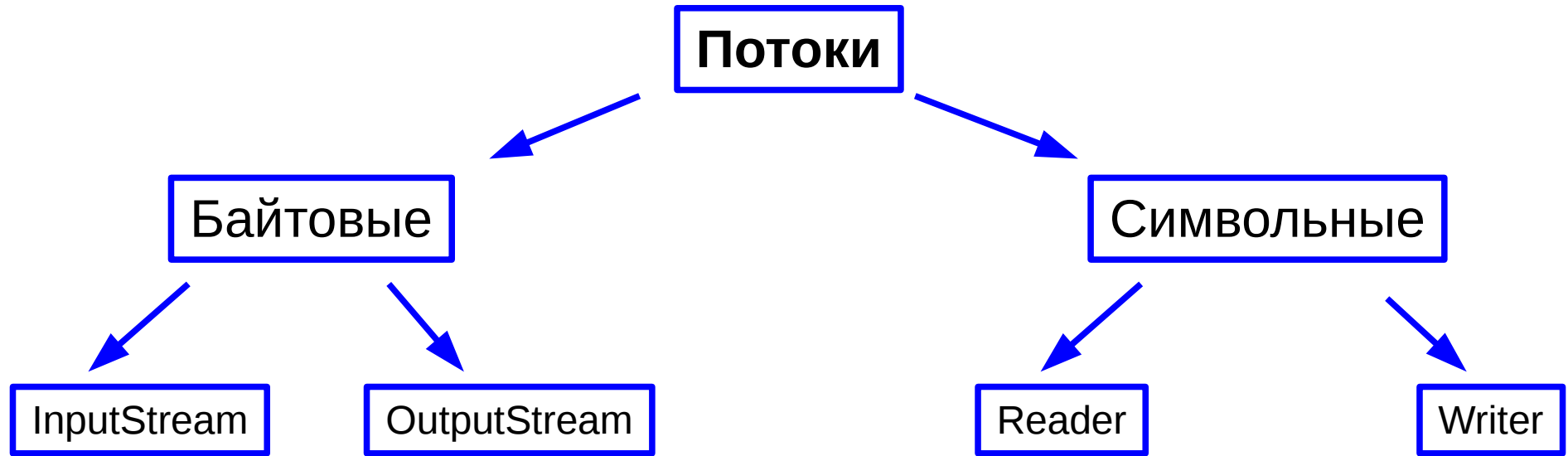


В результате в массив File попадут только файлы с указанным расширением



# Два типа потоков в Java

Потоки определены в пакете java.io



Все эти классы абстрактные

# Класс InputStream

Абстрактный класс, который определяет модель Java байтового потока ввода.

Метод	Описание
int available()	Возвращает количество байтов ввода, которые доступны в данный момент для чтения.
void close()	Закрывает источник ввода.
void mark( int кол-во_байтов)	Помещает метку в текущую точку входного потока, которая остается корректной до тех пор, пока не будет прочитано нужное кол-во байт.
boolean markSupported()	Возвращает true, если методы mark() и reset() поддерживаются вызывающим потоком.
int read()	Возвращает представление следующего доступного байта в потоке. При достижении конца потока возвращает значение -1.
int read(byte[] буфер)	Пытается читать некоторое кол-во байтов в буфер, возвращая количество успешно прочитанных байтов. При достижении конца потока возвращает значение -1.
int read(byte[] буфер, int смещение,int кол-во_байтов)	Действие аналогично предыдущему методу, однако байты записываются в буфер со смещением.
void reset()	Сбрасывает входной указатель в ранее установленную метку.
long skip (long кол-во_байт)	Игнорирует кол-во байт ввода, возвращает кол-во действительно проигнорированных байтов.

# Класс OutputStream

Абстрактный класс, который определяет модель Java байтового потока вывода.

Метод	Описание
int close()	Закрывает выходной поток. Последующие попытки записи передадут IOException.
void flush()	Финализирует выходное состояние, очищая буфер вывода.
void write( int b)	Записывает один байт в выходной поток.
void write(byte[] буфер)	Записывает полный массив байт в выходной поток.
void write(byte[] буфер,int смещение, int кол-во_байт)	Записывает диапазон из кол-ва байт из массива буфер начиная с буфер [смещение].

# Реализация класса InputStream - ByteArrayInputStream

Этот класс использует байтовый массив в качестве источника данных

Конструктор класса

`ByteArrayInputStream(byte[] массив)`

`ByteArrayInputStream(byte[] массив, int начало, int конец)`

К источнику данных применимы все операции потока ввода



# Пример использования ByteArrayInputStream

```
package com.gmail.tsa;
```

```
import java.io.*; ← Подключение пакета io
```

```
public class Main {
```

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

```
        String str="Hello world";  
        byte[] mas=str.getBytes();
```

Создание ByteArrayInputStream

```
        ByteArrayInputStream b=new ByteArrayInputStream(mas);  
        int c;  
        for(;;){  
            if (c==-1) break;
```

```
            c=b.read();
```

Использование стандартного метода  
InputStream

```
            System.out.println((char)c+" "+c);
```

```
        }
```

```
    }
```

```
}
```

# Реализация класса OutputStream - ByteArrayOutputStream

ByteArrayOutputStream хранит внутри себя данные в массиве байт и автоматически расширяется при выводе байтов в поток.

Конструктор класса

ByteArrayOutputStream() - создает буфер в 32 байта  
ByteArrayOutputStream( int ко-во\_байт)

К источнику данных применимы все операции потока вывода

# Пример использования ByteArrayOutputStream

```
package com.gmail.tsa;
```

```
import java.io.*;
```

```
public class Main {
```

Создание объекта байтового потока вывода

```
    public static void main(String[] args) {  
        ByteArrayOutputStream b= new ByteArrayOutputStream();
```

```
        byte[] buf={'a'};
```

```
        try{
```

```
            b.write(buf);
```

```
            b.write('b');
```

```
            String str="HELLO";
```

```
            b.write(str.getBytes());
```

```
        }
```

```
        catch(IOException e){
```

```
            System.out.println("Error");
```

```
        }
```

```
        byte[] c=b.toByteArray();
```

```
        for(int i=0;i<c.length;i++){
```

```
            System.out.print((char)c[i]);
```

```
        }
```

```
    }
```

```
}
```

Запись данных в буфер всегда выполняется внутри блока try

Получение массива byte на основе заполненного потока

# Для работы с файлами чаще всего используют `java.io.FileInputStream` и `java.io.FileOutputStream`

`FileInputStream` создает объект класса `InputStream`, который можно использовать для чтения из файла.

## Конструктор класса

`FileInputStream( String путь_к_файлу)`  
`FileInputStream( File объект файл)`

`FileOutputStream` создает объект класса `OutputStream`, который можно использовать для записи в файл.

## Конструктор класса

`FileOutputStream( String путь_к_файлу)`  
`FileOutputStream( File объект файл)`  
`FileOutputStream( String путь_к_файлу, boolean добавить)` — файл открывается в режиме добавления данных  
`FileOutputStream( File объект файл, boolean добавить)` — файл открывается в режиме добавления данных

# Пример использования FileOutputStream

```
package com.gmail.tsa;
import java.io.*;
public class Main {
    public static void main(String[]arg){
        String str="HELLO WORLD";
        FileOutputStream f=null;
        byte[] b=str.getBytes();
        try{
            f=new FileOutputStream("a.txt");
            f.write(b);
        }
        catch(IOException e){
            System.out.println("I/O Error");
        }

        finally{
            try{
                f.close();
            }
            catch(IOException e){
                System.out.println("Error close file");
            }
        }
    }
}
```

Создание объекта типа FileOutputStream

Запись байтового массива в файл

Заккрытие файла после записи

# Блок try с ресурсами

Для закрытия потока можно, либо вызвать явно метод `close()`, либо использовать введенный в JDK 7 оператор `try` с ресурсами.

Объявление блока `try` с ресурсами

```
try (описание ресурса){  
    Использование ресурса  
}
```

- 1) Ресурсы, управляемые оператором `try`, должны быть объектами классов, реализующих интерфейс `AutoCloseable`;
- 2) Ресурс, объявленный в блоке `try`, является неявно финальным;
- 3) Можно управлять несколькими ресурсами, отделив каждый из них в объявлении точкой с запятой;

# Пример использования FileInputStream

```
package com.gmail.tsa;  
import java.io.*;
```

```
public class Main {
```

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

```
        try(FileInputStream f= new FileInputStream("a.txt")){
```

```
            System.out.println("available byte = "+f.available());
```

```
            byte[] b=new byte[f.available()];
```

```
            f.read(b);
```

```
            for(byte a:b){
```

```
                System.out.print((char)a);
```

```
            }
```

```
        }
```

```
        catch(IOException e){
```

```
            System.out.println("FILE READ ERROR");
```

```
        }
```

```
    }
```

```
}
```

Использование  
try с ресурсами



Сколько байт  
можно считать  
с потока

Считывание массива  
байт с потока

# О побайтовом копировании файлов

Для копирования любого типа файлов применяются байтовые потоки ввода-вывода.

Алгоритм копирования файла:

- 1) Создать объекты классов `FileInputStream` и `FileOutputStream` для копируемого файла и копии соответственно.
- 2) Объявить буферный массив `byte` размером, примерно, 1024 или больше байт (внимание не слишком большой), так как это может увеличить объем памяти, потребляемой вашим приложением.
- 3) В цикле сначала вычитывать этот массив из файла, потом записывать в копию, и так до конца файла.



# Пример программы для копирования файла

```
package com.gmail.tsa;
```

```
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;
```

Импорт дополнительных пакетов

```
public class CopyFile {
```

Использование try с ресурсами для двух потоков

```
public static void main(String[] args) {  
  try (FileInputStream fis = new FileInputStream("a.txt");
```

Входной поток для файла

```
  FileOutputStream fos = new FileOutputStream("acopy.txt")) {
```

Выходной поток для копии

```
    byte[] buffer = new byte[1024];  
    int byteread = 0;
```

Буферный массив

```
    for (; (byteread = fis.read(buffer)) > 0;) {  
        fos.write(buffer, 0, byteread);  
    }
```

Копирование байтового  
содержимого из файла в его копию

```
  } catch (IOException e) {  
      System.out.println(e);  
  }  
  
}
```

```
}
```

# Запись типизированных данных в потоки

## `java.io.DataInputStream`

`DataInputStream` — класс входного потока, реализующего ввод типизированных данных.

Конструктор класса - `DataInputStream(InputStream Входной поток)`.

К источнику данных применимы все операции потока ввода и методы по вводу типизированных данных.

## `java.io.DataOutputStream`

`DataOutputStream` — класс выходного потока, реализующего вывод типизированных данных.

Конструктор класса - `DataOutputStream(OutputStream Выходной поток)`.

К источнику данных применимы все операции потока вывода и методы по выводу типизированных данных.

## Некоторые методы класса DataOutputStream

Метод	Описание
void writeDouble( double значение)	Записывает в поток переменную типа double
void writeBoolean(boolean значение)	Записывает в поток переменную типа boolean
void writeInt(int значение)	Записывает в поток переменную типа int

## Некоторые методы класса DataInputStream

Методы	Описание
double readDouble()	Считывает из потока переменную типа double
boolean readBoolean()	Считывает из потока переменную типа boolean
int readInt()	Считывает из потока переменную типа int

# Пример создания и использования DataOutputStream

```
package com.gmail.tsa;  
import java.io.*;  
public class Main {
```

Создание DataOutputStream: в качестве входного потока используется FileOutputStream

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

```
        try(DataOutputStream f=new DataOutputStream(new  
            FileOutputStream("b.dat"))){
```

```
            f.writeInt(12);
```

Запись целого числа в файл

```
        }
```

```
        catch(IOException e){
```

```
            System.out.println("File write error");
```

```
        }
```

```
    }
```

```
}
```

Внимание! Не смущайтесь, если открыв файл в текстовом редакторе, вы не обнаружите в нем чисел. Там и должны быть странные символы.

# Пример создания и использования DataInputStream

```
package com.gmail.tsa;  
import java.io.*;  
public class Main {
```

Создание DataInputStream: в качестве входного потока используется FileInputStream

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

```
        try(DataInputStream f=new DataInputStream(new  
            FileInputStream("b.dat"))){
```

```
            System.out.println(f.readInt());
```

```
        }  
        catch(IOException e){  
            System.out.println("ERROR READ FILE");  
        }
```

Чтение целого числа из файла

```
    }
```

```
}
```

# RandomAccessFile

**java.io.RandomAccessFile** – произвольный доступ к файлам.  
Позволяет читать, писать и перемещаться в любую позицию файла.

## Конструктор класса

RandomAccessFile( File объект\_файла, String доступ)  
RandomAccessFile( String имя\_файла, String доступ)

Тип доступа:

- «r» - только чтение;
- «rw» - чтение и запись;
- «rws» - чтение и запись, выполняемые сразу;

Для установки положения указателя в файле используется метод  
seek (long кол\_во байт от начала файла)

# Символьные потоки — потоки, оперирующие символами, а не байтами

**Reader** — абстрактный класс, определяющий символьный потоковый ввод в Java

Метод	Описание
<code>void close()</code>	Закрывает поток.
<code>void mark(int кол_во_символов)</code>	Перемещает метку на кол-во символов.
<code>boolean markSupported()</code>	Проверка поддержки методов <code>mark()</code> , <code>reset()</code> .
<code>int read()</code>	Возвращает целочисленное значение символа во входном потоке.
<code>int read (char[] буфер)</code>	Читает несколько символов в буфер и возвращает количество успешно прочитанных символов.
<code>boolean ready()</code>	Вернет <code>true</code> , если следующий запрос не будет ожидать.
<code>void reset()</code>	Сбрасывает указатель ввода в ранее установленную позицию-метку.
<code>long skip (long кол_во символов)</code>	Пропускает кол-во символов ввода, возвращая количество действительно пропущенных символов.

## **Writer** — абстрактный класс, определяющий символьный потоковый вывод в Java

Метод	Описание
Writer append(char символ)	Добавляет символ в конец вызывающего потока.
Writer append( CharSequence символы)	Добавляет символы в конец вызывающего потока.
Writer append(CharSequence символы, int начало, int конец)	Добавляет диапазон символов.
abstract void close()	Закрывает вызывающий поток.
abstract void flush()	Финализирует выходное состояние.
void write (int символ)	Записывает символ .
void write (char[] буфер)	Записывает массив в выходной поток.
void write (String строка)	Записывает строку в поток.
void write (String строка, int смещение , int кол_во символов)	Записывает определенное кол-во символов из строки.
void write (char [] буфер, int смещение , int кол_во символов)	Записывает определенное кол-во символов из массива char.



# Для работы с файлами чаще всего используют `java.io.FileReader` и `java.io.FileWriter`

`FileReader` создает объект класса `Reader` который можно использовать для чтения символов из файла.

## Конструктор класса

`FileReader( String путь_к_файлу)`  
`FileReader( File объект файл)`

`FileWriter` создает объект класса `Writer` который можно использовать для записи символов в файла.

## Конструктор класса

`FileWriter( String путь_к_файлу)`  
`FileWriter( File объект файл)`  
`FileWriter( String путь_к_файлу, boolean добавить)` — файл открывается в режиме добавления данных  
`FileWriter( File объект файл, boolean добавить)` — файл открывается в режиме добавления данных

# Класс `BufferedWriter`

**`BufferedWriter`** — класс, определяющий буферизированный СИМВОЛЬНЫЙ ПОТОКОВЫЙ ВВОД в Java

Конструктор класса

`BufferedWriter(Writer Выходной_поток)`

`BufferedWriter(Writer Выходной_поток, int размер буфера)`

# Пример использования BufferedWriter для записи в файл

```
package com.gmail.tsa;  
import java.io.*;  
public class Main {
```

Создание объекта BufferedWriter на основе FileWriter

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

```
try(BufferedWriter f=new BufferedWriter(new  
FileWriter("d.txt"))){
```

```
f.write("Hello");  
f.newLine();  
f.write("World");
```

Использование методов класса BufferedWriter для записи символов в файл

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

```
}  
catch(IOException e){  
    System.out.println();  
}
```

```
}
```

```
}
```

# Класс `BufferedReader`

**`BufferedReader`** — класс, определяющий символьный потоковый вывод в Java

Конструктор класса

`BufferedReader(Reader Входной_поток)`

`BufferedReader(Reader Входной_поток, int размер буфера)`

# Пример использования BufferedReader для чтения из файла

```
package com.gmail.tsa;  
import java.io.*;  
public class Main {
```

Создание объекта BufferedReader  
на основе FileReader

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

```
try(BufferedReader f =new BufferedReader(new  
FileReader("d.txt"))){
```

```
String str="";  
for(;(str=f.readLine())!=null;)  
System.out.println(str);
```

Использование метода  
readLine

```
}  
catch(IOException e){  
System.out.println("ERROR");  
}
```

```
}  
}
```

Удобным классом для записи данных в файл является класс **PrintWriter**

Конструктор класса

PrintWriter(OutputStream выходной\_поток)  
PrintWriter(OutputStream выходной\_поток, boolean сброс\_при\_новой\_строке)  
PrintWriter(Writer выходной\_поток)  
PrintWriter(Writer выходной\_поток, boolean сброс\_при\_новой\_строке)

выходной\_поток — Определяет объект класса OutputStream, который примет вывод

PrintWriter(File выходной\_файл)  
PrintWriter(File выходной\_файл, String набор\_символов)  
PrintWriter(String имя\_файла)  
PrintWriter(String имя\_файла, String набор\_символов)

! Удобство заключается в возможности использования методов `print()` и `println()`

# Пример использования PrintWriter

```
package com.gmail.tsa;  
import java.io.*;  
public class Main {
```

Создание PrintWriter  
для записи в файл

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

```
        try(PrintWriter a=new PrintWriter("c.txt")){
```

```
            for(int i=0;i<10;i++){  
                a.println(i);
```

```
            }  
            a.println();
```

```
            a.println("Hello WORLD");
```

```
        }
```

```
        catch(FileNotFoundException e){
```

```
            System.out.println("ERROR FILE WRITE");
```

```
        }
```

```
    }
```

```
}
```

Использование методов для  
записи в файл

## Краткие итоги урока

Поток - абстрактная величина, описывающая процесс передачи данных от источника к приемнику.

В Java потоки разделяются на два подтипа:

- **Байтовые** - манипулируют последовательностью байт;
- **Символьные** — манипулируют последовательностью символов;

В основе потоков лежат абстрактные классы:

Для **байтовых**:

`InputStream`

`OutputStream`

Для **символьных**:

`Reader`

`Writer`

Так как классы которые, отвечают за взаимодействие с тем или иным источником данных реализуют эти абстрактные классы, то их использование не предоставляет труда, так как их методы, в основном, одинаковы.

В Java существует два пакета для работы с потоками данных: IO и NIO. IO, на данный момент, популярнее и чаще используется. Основой для работы с файловой системой в IO является класс `File`.

Также начиная с JDK 1.7 введен удобный оператор **try** с ресурсами.



## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 717 — 759
- Кей Хорстман, Гари Корнелл Библиотека профессионала Java Том 2.Расширенные средства . стр. 20-45.
- Брюс Эккель Философия Java. 4-е издание 2009 г. стр. 483 - 507

## Домашнее задание

1. Напишите программу, которая копирует файлы с заранее определенным расширением(например, только doc) из каталога источника в каталог приемник.
2. Напишите программу, которая примет на вход два текстовых файла, а вернет один. Содержимым этого файла должны быть слова, которые одновременно есть и в первом и во втором файле.
3. Усовершенствуйте класс, описывающий группу студентов, добавив возможность сохранения группы в файл.
4. Реализовать обратный процесс. Т.е. считать данные о группе из файла, и на их основе создать объект типа группа.

# NIO 2

## Дополнительный материал к лекции «Потоки ввода-вывода»

**NIO** (Non-blocking I/O, New I/O) — коллекция прикладных программных интерфейсов для языка Java, предназначенных для реализации высокопроизводительных операций ввода-вывода. Также встречается аббревиатура NIO 2 – она относится к нововведениям относительно этого направления в Java 7.

NIO — коллекция прикладных программных интерфейсов для языка Java, предназначенных для реализации высокопроизводительных операций ввода - вывода. Также встречается аббревиатура NIO.2 – она относится к нововведениям относительно этого направления в Java 7.

#### Ключевые особенности NIO:

- Каналы и селекторы: NIO поддерживает различные типы каналов. Канал является абстракцией объектов более низкого уровня файловой системы (например, отображенные в памяти файлы и блокировки файлов), что позволяет передавать данные с более высокой скоростью. Каналы не блокируются и поэтому Java предоставляет еще такие инструменты, как селектор, который позволяет выбрать готовый канал для передачи данных, и сокет, который является инструментом для блокировки.
- Буферы: в Java 4 была введена буферизация для всех классов-обёрток примитивов (кроме Boolean). Появился абстрактный класс Buffer, который предоставляет такие операции, как clear, flip, mark и т.д. Его подклассы предоставляют методы для получения и установки данных.
- Кодировки: в Java 4 появились кодировки (java.nio.charset), кодеры и декодеры для отображения байт и символов Unicode.

# Использование NIO при работе с файловой системой

## Использование интерфейса Path

Java 7 представляет новую абстракцию для пути, а именно интерфейс Path. Он используется в новых функциях и API, по всему NIO.2. Объект пути содержит имена каталогов и файлов, которые составляют полный путь до файла/каталога, представленного объектом Path. Он содержит методы для извлечения элементов пути, манипуляций с ними и их добавления.

Один из классов, которые реализуют указанный интерфейс - это класс Paths, в котором есть статические методы для получения пути по указанному адресу.

## Пример использования интерфейса Path

```
package com.gmail.tsa;
```

```
import java.io.File;  
import java.nio.file.Path;  
import java.nio.file.Paths;
```

```
public class Main {
```

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

```
        Path p = Paths.get("src");
```

```
        System.out.println(p.getNameCount());
```

```
        System.out.println(p.getParent());
```

```
        System.out.println(p.isAbsolute());
```

```
        File f = p.toFile();
```

```
        System.out.println(f.getAbsolutePath());
```

```
    }
```

```
}
```

Присвоение переменной интерфейса  
объекта его реализующего (возвращается  
статическим методом класса Paths)

Из скольких частей  
состоит адрес

Родительский элемент

Проверка, абсолютный  
или относительный адрес

Можно на основе адреса  
создать объект класса File

## Использование класса Files

Files (введен в Java 7, находится в пакете `java.nio.file`) используется для выполнения различных операций с файлами и каталогами. Files является служебным классом - это означает, что это `final` - класс с `private` - конструктором и содержит только статические методы.

Название метода	Описание
<code>boolean isSameFile(Path1, Path2)</code>	Проверит, ссылаются ли два пути на один и тот же файл.
<code>Path copy(Path source, Path target, CopyOption... options)</code>	Копирование файла <code>source</code> в файл <code>target</code> .
<code>Path move(Path source, Path target, CopyOption... options)</code>	Перемещение файла <code>source</code> в файл <code>target</code> .
<code>void delete(pathSource);</code>	Удаление файла по адресу <code>pathSource</code> .
<code>boolean isReadable(), isWriteable() и isExecutable()</code>	Проверка возможность чтения, записи и исполнения файлов.
<code>boolean isDirectory()</code>	Проверка, на что указывает адрес: если на каталог, то вернет <code>true</code> , если на файл, то <code>false</code> .
<code>boolean exists()</code>	Проверка существования такого файла на диске.

## Использование классов Path и Files для работы с файлами

```
package com.gmail.tsa;
```

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
```

```
public class Main {
```

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

```
        Path p1 = Paths.get("a.txt");
```

Описываем адрес файла с помощью Path

```
        try {
```

```
            Files.createFile(p1);
```

Создаем файл на диске с помощью Files

```
        } catch (IOException e) {
```

```
            System.out.println("File already exists");
```

```
        }
```

```
        Path p2 = Paths.get("b.txt");
```

Переименовываем файл на диске

```
        try {
```

```
            Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

```
        } catch (IOException e) {
```

```
            System.out.println(e);
```

```
        }
```

Удаляем файл на диске

```
        try {
```

```
            Files.deleteIfExists(p2);
```

Можно указать стандартные опции копирования и перемещения

```
        } catch (IOException e) {
```

```
            System.out.println(e);
```

```
        }
```

```
    }
```

```
}
```



Система ввода-вывода NIO построена на двух основополагающих элементах: буферах и каналах.

В буфере хранятся данные.

Канал предоставляет открытое соединение с устройством ввода-вывода.

Для применения системы ввода-вывода NIO требуется получить канал для устройства ввода-вывода и буфер для хранения данных. После этого можно обращаться с буфером, вводя или выводя данные по мере необходимости.

**Буферы** определяются в пакете `java.nio`. Все буферы являются подклассами, производными от класса **Buffer**, в котором определяются основные функциональные возможности, характерные для каждого буфера, в том числе **текущая позиция**, **предел** и **емкость**. **Текущая позиция** определяет индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. **Предел** определяет значение индекса за позицией последней доступной ячейки в буфере. **Емкость** определяет количество элементов, которые можно хранить в буфере. Зачастую предел равен емкости буфера. В классе `Buffer` поддерживается также отметка и очистка буфера.

**Каналы** определены в пакете `java.nio.channels`. **Каналы** представляют открытое соединение с источником или адресатом ввода-вывода. Классы каналов реализуют интерфейс `Channel`.

## Методы класса Buffer

Метод	Описание
Object array()	Вернет ссылку на массив (если буфер поддерживает массив).
int arrayOffset()	Возвращает индекс первого элемента массива, если вызывающий буфер поддерживается массивом.
int capacity()	Возвращает количество элементов, которые можно хранить в вызывающем буфере.
Buffer clear()	Очищает вызывающий буфер и возвращает ссылку на него.
Buffer flip()	Задаёт текущую позицию в качестве предела для вызывающего буфера и затем устанавливает текущую позицию в нуль.
boolean hasArray()	Возвращает логическое значение true, если вызывающий буфер поддерживается массивом, доступным для чтения и записи, а иначе - логическое значение false.
boolean hasRemaining( )	Возвращает логическое значение true, если в вызывающем буфере ещё остались какие-нибудь элементы, а иначе - логическое значение false.
int limit()	Возвращает предел для вызывающего буфера.
Buffer limit(int n)	Задаёт предел n для вызывающего буфера. Возвращает ссылку на буфер.
int remaining()	Возвращает количество элементов, доступных до того, как будет достигнут предел.
Buffer reset()	Устанавливает текущую позицию в вызывающем буфере на установленной ранее метке.
Buffer rewind()	Устанавливает текущую позицию в вызывающем буфере в нуль.

## Методы класса ByteBuffer — предназначен для работы с байтовыми каналами

Метод	Описание
byte get()	Возвращает байт на текущей позиции.
ByteBuffer get (byte значения [ ] )	Копирует вызывающий буфер в заданный массив значения Возвращает ссылку на буфер.
ByteBuffer get (byte значения [ ] , int начало, int количество)	Копирует заданное количество элементов из вызывающего буфера в указанный массив значения, начиная с позиции по индексу начало. Возвращает ссылку на буфер.
byte get ( int индекс)	Возвращает из вызывающего буфера байт по указанному индексу.
ByteBuffer put (byte b)	Копирует заданный байт b на текущую позицию в вызывающем буфере. Возвращает ссылку на буфер.
ByteBuffer put (byte значения [ ] , int начало , int количество)	Копирует в вызывающий буфер заданное количество элементов из указанного массива значения, начиная с указанной позиции начало. Возвращает ссылку на буфер.
ByteBuffer put (ByteBuffer bb)	Копирует элементы из заданного буфера bb в вызывающий буфер, начиная с текущей позиции.
ByteBuffer put ( int индекс, byte b)	Копирует байт b на позицию по указанному индексу в вызывающем буфере.
ByteBuffer allocate(long n)	Выделить в ОЗУ место под n-байт для буфера.

## Каналы

Получения канала подразумевает вызов метода `getChannel()` для объекта, поддерживающего каналы. Конкретный тип возвращаемого канала зависит от типа объекта, для которого вызывается метод `getChannel()` .

Метод	Описание
<code>int read (ByteBuffer bb)</code>	Считывает байты из вызывающего канала в указанный буфер <code>bb</code> . Возвращает количество прочитанных байтов.
<code>int read (ByteBuffer bb, long start )</code>	Считывает байты из вызывающего канала в указанный буфер, начиная с позиции <code>start</code> . Текущая позиция не изменяется. Возвращает количество прочитанных байтов.
<code>int write (ByteBuffer bb )</code>	Записывает содержимое байтового буфера в вызывающий канал, начиная с текущей позиции. Возвращает количество записанных байтов.
<code>int write (ByteBuffer bb, long start)</code>	Записывает содержимое байтового буфера в вызывающий канал, начиная с позиции <code>start</code> в буфере. Текущая позиция не изменяется. Возвращает количество записанных байтов.

**package** com.gmail.tsa;

## Пример чтения данных с помощью системы NIO

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SeekableByteChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
```

```
public class Main {
```

```
    public static void main(String[] args) {
        Path pr = Paths.get("a.txt");
        try (SeekableByteChannel sbc = Files.newByteChannel(pr, StandardOpenOption.READ))
        {
```

```
            ByteBuffer bb = ByteBuffer.allocate(1024);
```

```
            int bufferread = 0;
```

```
            for (; (bufferread = sbc.read(bb)) > 0;) {
```

```
                bb.rewind();
```

```
                for (int i = 0; i < bufferread; i++) {
```

```
                    System.out.print((char) bb.get());
```

```
                }
```

```
            } catch (IOException e) {
```

```
                System.out.println(e);
```

```
            }
```

```
        }
```

```
    }
```

Открытие канала из файла

Создание объекта Path,  
указывающего на файл

Выделение в памяти  
под буфер 1024 байта

Установка указателя в начало  
буфера для чтения из буфера

Чтение по одному байту

## Пример ассоциации файла с буфером для записи данных

```
package com.gmail.tsa;
```

```
import java.io.IOException;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
```

```
public class Main {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Path p = Paths.get("a.txt");
```

```
        try (FileChannel fch = (FileChannel) Files.newByteChannel(p,
            StandardOpenOption.WRITE, StandardOpenOption.CREATE, StandardOpenOption.READ)) {
```

```
            MappedByteBuffer mbf = fch.map(FileChannel.MapMode.READ_WRITE, 0, 15);
```

```
            mbf.put("Hello word".getBytes());
```

```
        } catch (IOException e) {
            System.out.println(e);
```

```
        }
        System.out.println("Done");
    }
```

```
}
```

Создание канала присоединенного к файлу

Создание объекта Path, указывающего на файл

Запись данных в файл

Ассоциация буфера и файла

## Асинхронные операции ввода-вывода

Асинхронный ввод-вывод — это просто разновидность обработки ввода-вывода, при которой, еще до завершения записи или считывания, может выполняться другое действие.

В Java 7 появилось три новых асинхронных канала, на которые следует обратить внимание:

- `AsynchronousFileChannel` — для файлового ввода-вывода;
- `AsynchronousSocketChannel` — для сокетного ввода-вывода, поддерживает задержки;
- `AsynchronousServerSocketChannel` — для асинхронных сокетов, принимающих соединения;

Существуют две основные парадигмы (стиля) использования новых API асинхронного ввода-вывода: парадигма Future (с ожиданием) и парадигма Callback (обратный вызов).



## Стиль с ожиданием

Стиль с ожиданием (Future style) — термин, предложенный самими разработчиками API NIO.2. Он указывает на использование интерфейса `java.util.concurrent.Future`.

При стиле с ожиданием используется `java.util.concurrent`.

В таком случае объявляется объект `Future`, в котором будет храниться результат вашей асинхронной операции. Это означает, что работа актуального потока не будет тормозиться из-за потенциально медленной операции ввода-вывода. Напротив, операция ввода-вывода инициируется в отдельном потоке. По завершении операции возвращается ее результат. Тем временем ваш основной поток может и далее заниматься выполнением других задач, если это необходимо.



```
package com.gmail.tsa;
```

## Пример асинхронного потока с ожиданием

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
```

```
public class Main {
```

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

```
        Path file = Paths.get("a.txt");
```

Создание объекта Path,  
указывающего на файл

Открытие асинхронного канала

```
        try (AsynchronousFileChannel afc = AsynchronousFileChannel.open(file)) {
```

```
            ByteBuffer bb = ByteBuffer.allocate(10_000);
```

Буфер на 10 000 байт

```
            Future<Integer> result = afc.read(bb, 0);
```

```
            Integer byteRead = result.get();
```

```
            System.out.println(byteRead);
```

Передаем результат считывания в Future

```
        } catch (IOException | InterruptedException | ExecutionException e) {
            System.out.println(e);
        }
```

Считывание неявно начнется в параллельном потоке. Метод get() гарантированно дождетс окончания чтения.

```
    }
```

## Стиль с применением обратных вызовов

Главная идея заключается в том, чтобы основной поток отправил запрос (обработчик завершения `CompletionHandler` ) в отдельный поток, выполняющий операцию ввода-вывода. Запрос получит результат операции ввода-вывода, на основании которого будет запущен собственный метод этого запроса `completed` или `failed` (переопределяемый вами). После этого запрос вернется к основному потоку.

Интерфейс `java.nio.channels.CompletionHandler<V, A>` (где `V` — тип результата, а `A` — прикрепляемый объект, от которого вы получаете результат) активизируется после того, как будет выполнена асинхронная операция, связанная с вводом-выводом. Методы `completed(V, A)` и `failed(V, A)` этого интерфейса должны быть реализованы так, чтобы ваша программа четко различала случаи успешного и неуспешного завершения ввода-вывода и правильно на них реагировала.

## Пример асинхронного потока с обратным вызовом

```
package com.gmail.tsa;
```

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.channels.CompletionHandler;
import java.nio.file.Path;
import java.nio.file.Paths;
```

```
public class Main {
```

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

```
        Path file = Paths.get("a.txt");
```

```
        try (AsynchronousFileChannel afc = AsynchronousFileChannel.open(file)) {
            ByteBuffer bb = ByteBuffer.allocate(10_000);
```

```
            afc.read(bb, 0, bb, new CompletionHandler<Integer, ByteBuffer>() {
                @Override
                public void completed(Integer result, ByteBuffer attachment) {
                    System.out.println("Read OK");
                    System.out.println(new String(bb.array(), 0, result));
                }
            });
```

```
            @Override
            public void failed(Throwable exc, ByteBuffer attachment) {
                System.out.println("Read BAD");
            }
        });
```

```
        System.out.println(new String(bb.array()));
```

```
    } catch (IOException e) {
        System.out.println(e);
    }
```

```
}
```

Создание объекта Path,  
указывающего на файл

Открытие асинхронного канала

Анонимная реализация CompletionHandler

## Итоги урока

**NIO. 2** — представляет собой альтернативную реализацию процесса ввода-вывода в Java.

Для работы с файловой системой используется интерфейс **Path** (для указания адреса объекта файловой системы). Для работы с файловыми операциями используется сервисный класс **Files**, статические методы которого реализуют основные операции работы с файлами (создание, копирование, перемещение, удаление и т. д.).

Для реализации операций ввода-вывода используются такие абстракции как **каналы** — по сути способ чтения и записи данных, и **буферы** — временные хранилища данных.

Одним из основных преимуществ данного пакета является то, что операции записи и чтения неблокирующие, т. е., асинхронные. Это дает возможность одновременного считывания и записи данных из одного объекта, что может дать существенный прирост производительности при многопоточном выполнении программ.

## Дополнительная литература по теме данного урока.

- Бенджамин Эванс, Мартин Вербург Java новое поколение разработки, «Питер» - 2014 г, стр. 60-95
- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 641 - 474

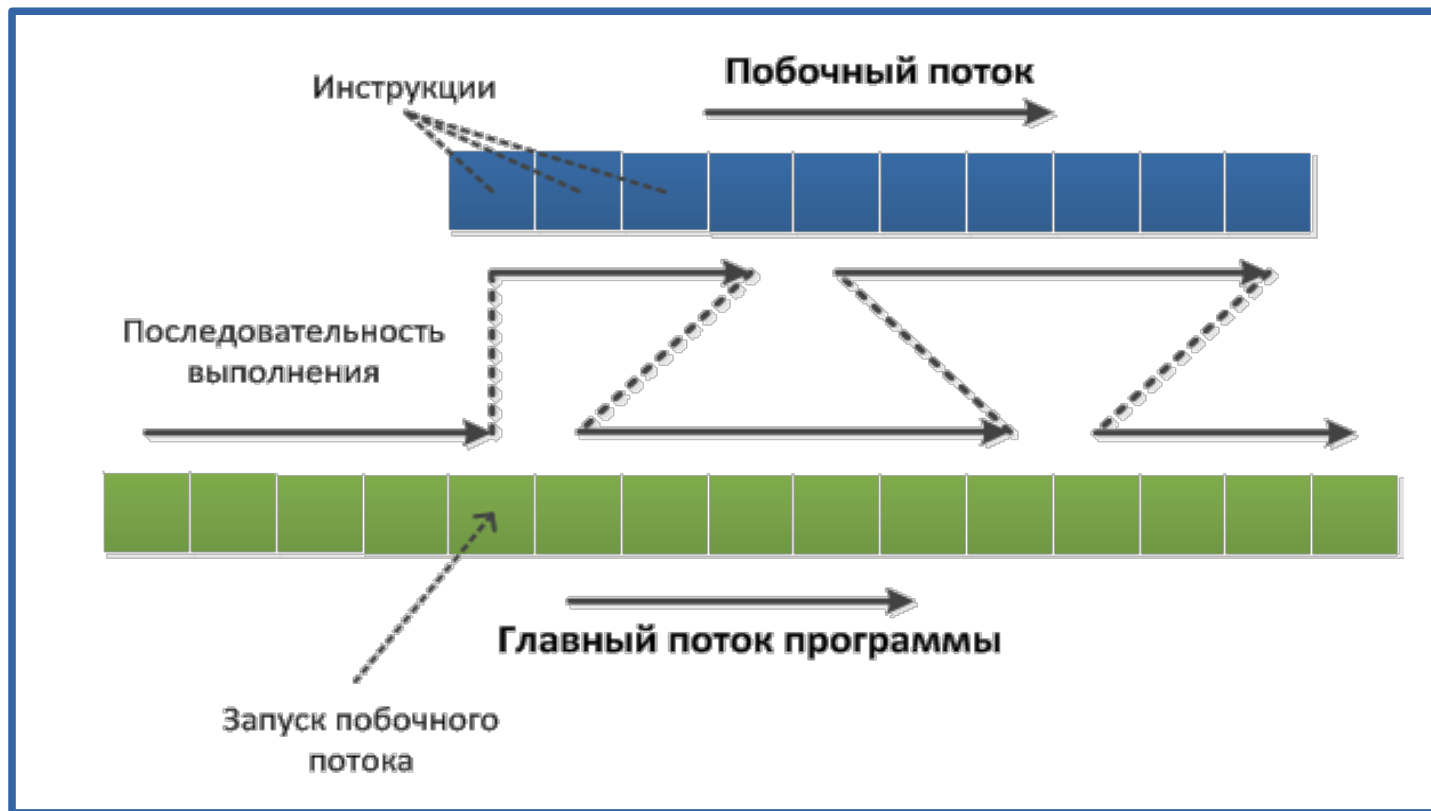
# Java OOP

## «Многопоточное программирование. Часть 1»

Процесс, порожденный в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть, без предписанного порядка во времени.

При выполнении некоторых задач такое разделение помогает достичь более эффективного использования ресурсов вычислительной машины.

# Схематическое изображение многопоточного выполнения программы



**!** В Java программа завершается тогда, когда завершаются все потоки, даже если они второстепенные. Однако, если дополнительный поток объявлен как демон, то он принудительно завершится после завершения основных.

## Оценка прироста производительности при использовании параллелизма

### Закон Амдала

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

$\alpha$  - Доля вычислений, которые могут быть выполнены только последовательно

$P$  - количество «вычислителей»

$S_p$  - прирост производительности

### Закон Густавсона — Барсиса

$$S_p = p + (1 - p)g$$

$g$  — доля последовательных расчетов в программе

$p$  - количество процессоров

$S_p$  - прирост производительности



# В Java многопоточность реализована на основе класса Thread и его интерфейса Runnable

Поток инкапсулируется классом Thread . Для создания нового потока нужно, либо расширить класс Thread, либо реализовать интерфейс Runnable

Методы класса Thread

Метод	Назначение
String getName()	Получить имя потока.
int getPriority()	Получить приоритет потока.
boolean isAlive()	Определить, выполняется ли поток.
void join()	Ожидать завершение потока.
void run()	Входная точка потока.
void sleep(long time)	Приостановить поток на заданное время.
void start()	Запустить поток вызовом его метода run.

# Главный поток

При старте любой Java программы начинает выполняться главный поток. Для управления можно использовать объект Thread. Ссылку на него можно получить, используя метод `currentThread()`

Общий формат вызова - `static Thread currentThread()`

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Thread t=Thread.currentThread();
```

```
        System.out.println(t.getName());
```

```
        try{
```

```
            t.sleep(3000);
```

```
        }
```

```
        catch(InterruptedException e){
```

```
            System.out.println("main thread break");
```

```
        }
```

```
        System.out.println("Stop programm");
```

```
    }
```

```
}
```

Получение главного потока

Остановка потока на 3 сек

# Создание потока реализацией интерфейса Runnable

Можно создать новый поток из любого объекта, реализующего интерфейс Runnable.

Для реализации этого интерфейса достаточно реализовать один метод - **public void run()**.

Код, объявленный внутри метода run(), и является кодом нового потока. Поток завершится, когда метод run() вернет управление.

После этого нужно создать объект типа Thread из этого класса. Для этого используют конструктор вида

Thread (Runnable **объект\_потока**, String имя\_потока)

**объект\_потока** объект класса, реализующего интерфейс Runnable

После того, как поток создан, его нужно запустить методом start().

# Создание объекта реализацией интерфейса Runnable

## Пример

```
package com.gmail.tsa;

public class NewThread implements Runnable{

    @Override
    public void run(){ ← Реализация интерфейса Runnable

        for (int i=0;i<160;i++){
            System.out.println(Thread.currentThread()
                               .getName()+" - "+i);
        }
    }
}
```

Т.е., в параллельном потоке будет просто выводиться имя этого потока и числа от 0 до 160.

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Thread thr1=new Thread(new NewThread(),"2 thr");  
        Thread thr2=new Thread(new NewThread(),"3 thr");  
        Thread thr3=new Thread(new NewThread(),"4 thr");
```

```
        thr1.start();  
        thr2.start();  
        thr3.start();
```

Старт всех потоков

```
        System.out.println("Stop programm");
```

```
    }
```

```
}
```

Последний оператор главного метода. Однако несмотря на то, что главный метод закончится параллельные потоки будут продолжать свою работу.

Создание потоков на основе класса реализующего интерфейс Runnable

# Создание нового потока расширением класса Thread

Алгоритм создания нового потока:

- 1) Создается класс, который наследует класс Thread.
- 2) Переопределяется метод run().
- 3) При необходимости переопределяются иные методы.
- 4) Создается объект на основе класса.
- 5) Запускается поток используя метод start().

# Пример класса расширяющего класс Thread

```
package com.gmail.tsa;
```

```
public class NewThread2 extends Thread{  
    NewThread2(String g){  
        super(g);  
    }  
}
```

↑  
Наследование класса Thread

```
@Override
```

```
public void run(){  
    for(int i=0;i<10;i++){  
        System.out.println(Thread.currentThread()  
            .getName()+" - "+i);  
    }  
}
```

← Переопределение метода run()

## Пример создания новых потоков на основе переопределения Thread

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Thread thr1=new Thread(new NewThread(),"2 thr");
```

```
        Thread thr2=new Thread(new NewThread(),"3 thr");
```

```
        Thread thr3=new Thread(new NewThread(),"4 thr");
```

```
        NewThread2 thr4=new NewThread2("5 thr");
```

```
        NewThread2 thr5=new NewThread2("6 thr");
```

```
        NewThread2 thr6=new NewThread2("7 thr");
```

```
        thr1.start();
```

```
        thr2.start();
```

```
        thr3.start();
```

```
        thr4.start();
```

```
        thr5.start();
```

```
        thr6.start();
```

```
        System.out.println("Stop programm");
```

```
    }
```

```
}
```

Создание объектов  
переопределенного  
класса

Запуск созданных потоков



## Какой из этих двух путей выбрать ?

! В общем случае оба подхода равноправны.

Рекомендуется (но не обязательно !) использовать реализацию интерфейса `Runnable`, если, кроме метода `run()`, ничего более не переопределяется.

## Состояния потоков

- **NEW** – поток создан, но еще не запущен;
- **RUNNABLE** – поток выполняется;
- **BLOCKED** – поток блокирован;
- **WAITING** – поток ждет окончания работы другого потока;
- **TIMED\_WAITING** – поток некоторое время ждет окончания другого потока;
- **TERMINATED** — поток завершен;

! Состояние потока можно получить используя метод переменной потока `getState()`.

# Приоритет потоков

Каждый поток обладает приоритетом выполнения. Чем выше приоритет, тем больше процессорного времени ему выделяется.

Установить приоритет выполнения потока можно с помощью функции

```
void setPriority(int уровень)
```

Уровень - число от MIN\_PRIORITY до MAX\_PRIORITY — 0..10

Получить приоритет потока      int **getPriority**()

Внимание! Установка одинакового приоритета не гарантирует одинаковой скорости выполнения потоков.

# Методы для работы с потоками

Имя метода	Описание
<code>boolean isAlive()</code>	Вернет true, если поток ,вызвавший этот метод, еще выполняется.
<code>void join()</code>	Ожидает завершения потока, для которого он вызван.
<code>void setDaemon(boolean)</code>	Устанавливает поток демоном.
<code>void interrupt()</code>	Устанавливает флаг прерывания потока в true. Т.е., дает указание на прерывание потока.
<code>void yield()</code>	Дает указание планировщику выполнить другой поток, который ожидает своей очереди.

## Пример использования join()

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Thread thr1=new Thread(new NewThread(),"2 thr");
```

```
        NewThread2 thr4=new NewThread2("5 thr");
```

```
        try{
```

```
            thr1.start();
```

```
            thr4.start();
```

```
            thr1.join();
```

```
            thr4.join();
```

```
        }
```

```
        catch (InterruptedException e){
```

```
        }
```

```
        System.out.println("Stop programm");
```

```
    }
```

```
}
```

Теперь главный поток ждет окончания работы дополнительных потоков.

# Пример использования setDaemon, isDaemon, interrupt

```
package com.gmail.tsa;
```

```
public class Main {
```

Получение главного потока

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

```
        Thread t=Thread.currentThread();
```

```
        Thread thr1=new Thread(new NewThread(),"2 thr");
```

```
        thr1.setDaemon(true);
```

Установка потока демоном

```
        System.out.println(thr1.isDaemon());
```

```
        thr1.start();
```

```
        System.out.println("Stop programm");
```

```
        t.interrupt();
```

Прерывание основного потока

```
    }
```

```
}
```

## Пример ускорения операций с использованием МНОГОПОТОЧНОСТИ

```
package com.gmail.tsa;
```

```
public class SingleThreadSorting implements Runnable {  
    private int[] array;  
    private int begin;  
    private int end;  
    private Thread thr;  
    private int index;  
    private boolean stop = false;
```

Класс реализующий Runnable



```
    public SingleThreadSorting(int[] array, int begin, int end) {  
        super();  
        this.array = array;  
        this.begin = begin;  
        this.end = end;  
        thr = new Thread(this);  
        thr.start();  
        this.index = begin;  
    }
```


```
    public Thread getThr() {  
        return thr;  
    }
```

```
    public int peekElement() {  
        return array[index];  
    }
```

```
    public int pollElement() {  
        int temp = array[index];  
        check();  
        return temp;  
    }
```

```
    public boolean isStop() {  
        return stop;  
    }
```

Конструктор, получающий массив по ссылке, и также создающий и запускающий поток.



class SingleThreadSorting

## Пример ускорения операций с использованием многопоточности (продолжение)

@Override

```
public void run() {  
    int temp;  
    for (int i = begin; i < end; i++) {  
        int k = i - 1;  
        temp = array[i];  
        for (; k >= begin && array[k] > temp;) {  
            array[k + 1] = array[k];  
            array[k] = temp;  
            k--;  
        }  
    }  
}
```

Сортировка массива вставкой

```
private void check() {  
    this.index++;  
    if (this.index >= this.end) {  
        this.stop = true;  
    }  
}
```

Вспомогательный метод

class SingleThreadSorting

Теперь нужно объединить несколько потоков для сортировки всего массива, а потом собрать полученные элементы в один массив.



## Класс, формирующий заданное количество потоков, и запускающий их на выполнение

```
package com.gmail.tsa;
```

```
public class MultiThreadSorting {  
    static void sort(int[] array, int threadNumber) {  
        SingleThreadSorting[] threadarray = new SingleThreadSorting[threadNumber];  
        for (int i = 0; i < threadarray.length; i++) {  
            int size = array.length / threadNumber;  
            int begin = size * i;  
            int end = ((i + 1) * size);  
            if ((array.length - end) < size) {  
                end = array.length;  
            }  
            threadarray[i] = new SingleThreadSorting(array, begin, end);  
        }  
        for (int i = 0; i < threadarray.length; i++) {  
            try {  
                threadarray[i].getThr().join();  
            } catch (InterruptedException e) {  
                System.out.println(e);  
            }  
        }  
        System.arraycopy(mergeArrays(array, threadarray), 0, array, 0, array.length);  
    }  
}
```

Создаем массив потоков и иницилируем их

Вызов метода для слияния получившихся частей

class MultiThreadSorting

## Класс, формирующий заданное количество Потоков, и запускающий их на выполнение

```
private static int[] mergeArrays(int[] array, SingleThreadSorting[] threadarray) {  
    int[] arr = new int[array.length];  
    for (int i = 0; i < arr.length; i++) {  
        int min = Integer.MAX_VALUE;  
        int k = -1;  
        for (int j = 0; j < threadarray.length; j++) {  
            if (!threadarray[j].isStop() && min > threadarray[j].peekElement()) {  
                min = threadarray[j].peekElement();  
                k = j;  
            }  
        }  
        if (k != -1) {  
            arr[i] = threadarray[k].pollElement();  
        }  
    }  
    return arr;  
}
```

Метод для слияния частей массива из потоков

class MultiThreadSorting

## Главный класс для тестирования ускорения при использовании многопоточности

```
package com.gmail.tsa;

import java.util.Arrays;
import java.util.Random;

public class Main {

    public static void main(String[] args) {
        int[] array = new int[200000];
        Random rn = new Random();
        for (int i = 0; i < array.length; i++) {
            array[i] = rn.nextInt(10);
        }
        int[] array2 = array.clone();
        int[] array3 = array.clone();
        long tstart = System.currentTimeMillis();
        sort(array);
        long tend = System.currentTimeMillis();
        System.out.println((tend - tstart) + " ms" + "- Static method sort");
        tstart = System.currentTimeMillis();
        MultiThreadSorting.sort(array2, 3);
        tend = System.currentTimeMillis();
        System.out.println((tend - tstart) + " ms" + "- MultiThread sort");
        tstart = System.currentTimeMillis();
        Arrays.sort(array3);
        tend = System.currentTimeMillis();
        System.out.println((tend - tstart) + " ms" + "- Array sort");
    }

    static void sort(int[] array) {
        int temp;
        for (int i = 1; i < array.length; i++) {
            int k = i - 1;
            temp = array[i];
            for (; k >= 0 && array[k] > temp; k--) {
                array[k + 1] = array[k];
                array[k] = temp;
            }
        }
    }
}
```

Создание трех массивов  
наполненных случайными числами

Запуск сортировки в  
разных режимах для  
сравнения скорости

Статический метод для  
однопоточной  
сортировки

## Итоги урока

Поток - это часть процесса вашей программы, в которой могут производиться параллельные вычисления. Использование многопоточного программирования позволяет наиболее полно использовать ресурсы современных ПК.

В Java поток инкапсулируется классом **Thread**. Создать новый поток можно двумя способами:

- Реализуя интерфейс **Runnable**. И создав Thread на основе этого класса;
- Наследуясь от **Thread**;

И в первом и во втором случае следует переопределить метод **public void run()**. Все что описано в этом методе выполниться в параллельном потоке.

И в первом и во втором случае созданный поток запускается на выполнение вызовом метода **start()**.

Приложение считается завершившим работу, если завершились все порожденные им потоки. Необработанное исключение аварийно завершает только тот поток в котором, было сгенерировано. На работу остальных потоков оно не влияет.

Можно присоединить один поток к другому (второй будет ждать окончания работы первого) с помощью метода **join()**.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 259 — 271
- Кей Хорстманн, Гари Корнелл Библиотека профессионала. Java том -1. стр.753-772

## Домашнее задание

- 1) Создайте сто потоков, которые будут вычислять факториал числа, равного номеру этого потока, и выводить результат на экран.
- 2) Написать код для многопоточного подсчета суммы элементов массива целых чисел. Сравнить скорость подсчета с простым алгоритмом.
- 3) Напишите многопоточный вариант сортировки массива методом Шелла.
- 4) Реализуйте многопоточное копирование каталога, содержащего несколько файлов.
- 5) Реализуйте программу, которая с периодичностью 1 сек, будет проверять состояние заданного каталога. Если в этом каталоге появится новый файл или исчезнет существующий, то выведется сообщение об этом событии. Программа должна работать в параллельном потоке выполнения.

# Executors и интерфейсы Callable и Future Дополнительный материал к лекции «Многопоточное программирование. Часть 1 »

Исполнители (Executors) и интерфейсы Callable и Future - предоставляют более удобные средства для многопоточного программирования в сравнении с базовой реализацией на основе класса Thread.

Составил: Цымбалюк А.Н.

## Интерфейс Callable

Интерфейс Callable инкапсулирует асинхронный параллельный поток выполнения, который может возвращать значение. Это - обобщенный интерфейс. Метод для его переопределения - call().

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

Параметр типа V описывает тип возвращаемого значения. Так, например, Callable<Integer> означает, что метод call вернет переменную типа Integer.

**Для сохранения результатов таких вычислений чаще всего используется интерфейс Future<?>**



## Интерфейс Future

Интерфейс Future используется для получения результатов из параллельного потока для использования в дальнейшем.

```
public interface Future<V> {  
    V get() throws Exception;  
  
    V get(long timeout, TimeUnit unit) throws Exception;  
  
    void cancel(boolean mayInterrupt);  
  
    boolean isCancelled();  
  
    boolean isDone();  
}
```

Параметр типа V — описывает тип возвращаемого значения.

**Основная особенность интерфейса - можно получить результат из параллельного потока, не заботясь об его окончании. Результат будет использован только тогда когда его поток гарантированно завершится.**

## Методы интерфейса Future

Метод	Описание
<code>V get()</code>	Установит блокировку в вызывающем потоке до тех пор пока, не получит результата вычислений.
<code>V get(long timeOut, TimeUnit unit)</code>	Установит блокировку в вызывающем потоке до получения результата вычислений и, если закончится время заданное в параметрах метода то сгенерирует исключение.
<code>void cancel()</code>	Прервать текущие вычисления.
<code>boolean isCanceled()</code>	Вернет true, если вычисления отложены.
<code>boolean isDone()</code>	Вернет true если, вычисления закончены.

## Класс-оболочка FutureTask<?>

Служит для создания на основе интерфейса Callable двух интерфейсов Future и Runnable, реализуя оба этих интерфейса.

Этот класс обобщенный, и в качестве параметра конструктора должен выступать объект класса, реализующего Callable.

Для создания Thread на его основе, просто передайте экземпляр класса FutureTask конструктору Thread. Далее запустите поток на выполнение методом start().

## Пример класса реализующего интерфейс Callable

```
package com.gmail.tsa;
```

```
import java.io.File;
```

```
import java.util.concurrent.Callable;
```

Метод call() должен вернуть Long



```
public class FileLengtCounter implements Callable<Long> {  
    private String fileAdress;
```

```
    public FileLengtCounter(String fileAdress) {  
        super();  
        this.fileAdress = fileAdress;  
    }  
  
    @Override  
    public Long call() throws Exception {  
        File file = new File(fileAdress);  
        if (!file.exists() || !file.isFile()) {  
            return 0L;  
        }  
  
        return file.length();  
    }  
}
```

Реализация  
интерфейса Callable.  
Метод возвращает  
размер файла в  
байтах.



```
package com.gmail.tsa;
import java.io.File;
import java.util.ArrayList;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;
```

```
public class Main {
```

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

```
        System.out.println(calculateFolderSize("someFolder"));
```

```
    }
```

```
    public static Long calculateFolderSize(String folderAddress) {
```

```
        File file = new File(folderAddress);
```

```
        File[] fileArray = file.listFiles();
```

```
        ArrayList<Future<Long>> result = new ArrayList<>();
```

```
        for (File fileElement : fileArray) {
```

```
            FutureTask<Long> res = new FutureTask<>(new FileLengtCounter(fileElement.getAbsolutePath()));
            result.add(res);
```

```
            Thread thread = new Thread(res);
            thread.start();
```

```
        }
```

```
        Long totalSize = 0L;
```

```
        for (Future<Long> future : result) {
```

```
            try {
```

```
                totalSize += future.get();
```

```
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
```

```
        }
```

```
        return totalSize;
```

```
    }
```

```
}
```

Список для результата работы каждого потока

Создание FutureTask на основе интерфейса Callable

Создание и запуск потока на основе задачи

Получение результатов из параллельных потоков

Программа, которая в параллельных потоках вычислит размер файлов в каталоге, и вернет их сумму.

## Интерфейсы, предназначенные для более удобного запуска работы

Для более удобного управления потоками применяются интерфейсы, направленные на управление Runnable классами.

Иерархия интерфейсов и методов с ними связанных

**Executor**

Упрощение создания и запуска потока

**ExecutorService**

Возможность создания потокового пула

**ScheduledExecutorService**

Создание потокового пула с планировщиком по времени



# Executors

Большинство классов, реализующих данные интерфейсы, создаются с помощью статических методов-фабрик класса Executors.

Интерфейс реализуемый возвращаемым классом	Метод	Описание
ExecutorService	<code>newSingleThreadExecutor()</code>	Создает исполнителя на один поток.
ExecutorService	<code>newFixedThreadPool(int n)</code>	Создает пул на n потоков. Если на выполнение поставлено задач больше, чем потоков, то они ставятся в очередь.
ExecutorService	<code>newCachedThreadPool()</code>	Создает пул потоков, в котором новые потоки создаются по мере необходимости.
ScheduledExecutorService	<code>newSingleThreadScheduledExecutor()</code>	Пул из одного потока с поддержанием планировщика запуска.
ScheduledExecutorService	<code>newScheduledThreadPool(int n)</code>	Фиксированный потоковый пул с планировщиком запуска.

## Описание некоторых классов реализующих интерфейсы **Executor**, **ExecutorService**, **ScheduledExecutorService**

- **SingleThreadExecutor** – все задачи выполняются по очереди.  
**Executors.newSingleThreadExecutor()**.
- **CachedThreadPool** – пул создает столько потоков, сколько ему нужно для выполнения задач параллельно. Старые объекты-потоки повторно используются для новых задач. Если поток не используется более 60 сек, он будет остановлен и удален из пула.  
**Executors.newCachedThreadPool()**
- **FixedThreadPool** – пул с фиксированным количеством потоков. Если нет свободного потока, то задача попадает в очередь.  
**Executors.newFixedThreadPool(int n)**
- **ScheduledThreadPool** – пул с планировщиком.
- **Executors.newScheduledThreadPool(int n)**
- **Single Thread Scheduled Pool** – пул с планировщиков на один поток.  
**Executors.newSingleThreadScheduledExecutor()**



## Класс `FixedThreadPool`

**`FixedThreadPool`** — используется, когда существует множество задач, которые нужно запустить на параллельное выполнение, но, при этом, фиксировать количество одновременно выполняемых потоков. Это позволит сохранить производительность программы на приемлемом уровне.

Для этого нужно создать фиксированный пул потоков на нужное количество потоков. Потом запустить произвольное количество задач на этом пуле. Этот пул гарантирует, что одновременно будет работать только заданное количество потоков.

Внимание! Методы для запуска `Runnable` и `Callable` задач отличаются.

После окончания работы потокового пула следует вызвать метод `shutdown()` для выгрузки и завершения всех потоков.

## Методы для запуска потоков реализующих интерфейсы Runnable и Callable

Интерфейс	Метод
Runnable	Future<?> execute (Runnable task)
Runnable	Future<?> submit (Runnable task)
Runnable	Future<T> submit (Runnable task, T result)
Callable	Future<T> submit (Callable<T> task)

**Внимание! Для Runnable задач Future.get() вернет null для первого и второго метода, и то, что было вторым параметром для третьего метода.**

# Пример использования фиксированного потокового пула

```
public static Long calculateFolderSize(String folderAddress) {
```

```
    File file = new File(folderAddress);  
    File[] fileArray = file.listFiles();
```

Потоковый пул на три потока

```
    ExecutorService exSer = Executors.newFixedThreadPool(3);
```

```
    ArrayList<Future<Long>> result = new ArrayList<>();
```

```
    for (File fileElement : fileArray) {
```

```
        result.add(exSer.submit(new FileLengtCounter(fileElement.getAbsolutePath())));
```

```
    }  
    Long totalSize = 0L;
```

Запуск задач на вычисление

```
    for (Future<Long> future : result) {  
        try {
```

```
            totalSize += future.get();
```

Считывание полученных результатов

```
        } catch (InterruptedException | ExecutionException e) {  
            e.printStackTrace();
```

```
        }  
    }
```

```
    exSer.shutdown();
```

Завершение потокового пула

```
    return totalSize;
```

```
}
```

**Постановка задачи:** переписать метод по многопоточному подсчету суммы размеров файлов с использованием фиксированного пула потоков на три потока. Т.е. одновременно параллельно выполняется только три задачи.

## Методы интерфейса ExecutorService для работы с группой потоков

ExecutorService предлагает методы, которые могут манипулировать группами потоков, объединенных в классы-контейнеры (например список). Это позволит запустить на выполнение все Callable задачи, хранимые в этом списке, и получить список результатов.

Метод	Описание
T invokeAny(Collection<Callable<T>> tasks)	Вернет первый полученный результат из списка задач. Все остальные задачи останавливаются.
T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)	Вернет первый полученный результат из списка задач. Все остальные задачи останавливаются. Если закончится время заданное timeout сгенерируется исключение TimeoutException.
List<Future<T>> invokeAll(Collection<Callable<T>> tasks)	Вернет список полученных результатов вычисления задач.
List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)	Вернет список полученных результатов вычисления задач. Если закончится время заданное timeout, сгенерируется исключение TimeoutException.

# Пример использования групповой обработки задач

```
public static Long calculateFolderSize(String folderAddress) {  
    File file = new File(folderAddress);  
    File[] fileArray = file.listFiles();
```

Создание списка задач

```
    ExecutorService exSer = Executors.newFixedThreadPool(3);  
  
    ArrayList<Callable<Long>> tasks = new ArrayList<>();  
    for (File fileElement : fileArray) {  
        tasks.add(new FileLengthCounter(fileElement.getAbsolutePath()));  
    }
```

```
    List<Future<Long>> result = null;
```

Список для результатов

```
    try {  
        result = exSer.invokeAll(tasks);  
    } catch (InterruptedException e1) {  
        e1.printStackTrace();  
    }
```

Получение списка результатов,  
используя групповой запуск задач

```
    Long totalSize = 0L;  
    for (Future<Long> future : result) {  
        try {  
            totalSize += future.get();  
        } catch (InterruptedException | ExecutionException e) {  
            e.printStackTrace();  
        }  
    }  
  
    exSer.shutdown();  
    return totalSize;  
}
```

**Постановка задачи:** переписать метод по многопоточному подсчету суммы размеров файлов с использованием фиксированного пула потоков на три потока. Т.е. одновременно параллельно выполняется только три задачи. Добавить методы для работы с группой задач

## Класс ExecutorCompletionService

Недостатком указанной групповой политики обработки задач является то, что результат задач выбирается в порядке следования и для первых результатов придется ждать слишком долго. Для решения можно использовать класс, который дает возможность выбирать результаты, в зависимости от готовности - **ExecutorCompletionService**

Конструктор класса принимает на вход исполнитель который будет использоваться для запуска задач

Конструктор - ExecutorCompletionService (ExecutorService executor)

Метод	Описание
Future<T> submit(Callable<T> task)	Передаст задание исполнителю (который был параметром конструктора).
Future<T> submit(Runnable task, T result)	Передаст задание исполнителю (который был параметром конструктора).
Future<T> take()	Удаляет готовый результат и возвращает его или устанавливает блокировку, если их нет.
Future<T> poll()	Удаляет готовый элемент и возвращает его. Если готовых нет, то возвращает null.
Future<T> poll(long time, TimeUnit unit)	Удаляет готовый элемент и возвращает его. Если готовых нет, то возвращает null. Ожидает указанное время.

# Использование исполнителей с планировщиком

Исполнители с планировщиком описаны интерфейсом **ScheduledExecutorService**

Методы класса Executors, возвращающие классы, реализующие интерфейс **ScheduledExecutorService**:

- **newScheduledThreadPool()** - вернет потоковый пул с планировщиком;
- **newSingleThreadScheduledExecutor()** - вернет поток работающий с планировщиком;

## Методы **ScheduledExecutorService**

Метод	Описание
<code>ScheduledFuture&lt;T&gt; schedule (Callable&lt;T&gt; task,long time, TimeUnit unit)</code>	Запуск задачи произойдет через указанное время.
<code>ScheduledFuture&lt;?&gt; schedule (Runnable task,long time, TimeUnit unit)</code>	Запуск задачи произойдет через указанное время.
<code>ScheduledFuture&lt;?&gt; scheduleAtFixRaye (Runnable task,long init,long period, TimeUnit unit)</code>	Запускает задачу через period времени, после задержки init.
<code>ScheduledFuture&lt;?&gt; scheduleWithFixDelay(Runnable task,long init,long delay, TimeUnit unit)</code>	Планирует запуск задачи со временем задержки delay в единицах unit, между запусками после окончания стартового времени init.

## Перечисление TimeUnit

В параллельном API определяются методы, принимающие параметр перечисляемого типа TimeUnit, обозначающего период времени ожидания. Перечисление TimeUnit служит для обозначения степени разрешения синхронизации. Это перечисление определено в пакете `java.util.concurrent` и может принимать одно из следующих значений:

- DAYS
- HOURS
- MINUTES
- SECONDS
- MICROSECONDS
- MILLISECONDS
- NANOSECONDS

Также это перечисление снабжено методами, выполняющими перевод из одних единиц времени в другие.



## Методы TimeUnit

Метод	Описание
<code>long conver(long time, TimeUnit unit)</code>	Преобразует time в заданные единицы перечисления unit.
<code>long toMicros(long time)</code>	Преобразует time в микросекунды.
<code>long toMillis(long time)</code>	Преобразует time в миллисекунды.
<code>long toNanos(long time)</code>	Преобразует time в наносекунды.
<code>long toSeconds(long time)</code>	Преобразует time в секунды.
<code>long toDays(long time)</code>	Преобразует time в дни.
<code>long toHours(long time)</code>	Преобразует time в часы.
<code>long toMinutes(long time)</code>	Преобразует time в минуты.

Описанные выше методы используются для перевода единиц времени из одних в другие. Однако у перечисления существует ряд методов, направленных на решение задач синхронизации.

## Синхронизирующие методы TimeUnit

Это перечисление снабжено рядом методов для приостановления выполнения потока на определенный период времени.

Метод	Описание
<code>void sleep(long time)</code>	Приостанавливает поток на указанное время.
<code>void timeJoin(Thread thread, long time)</code>	Присоединяет активный поток к thread на указанное время time.
<code>void timeWait(Object object, long time)</code>	Переводит поток, для которого этот метод вызван на указанное время, в состояние wait.

Внимание! Все эти методы помечены как генерирующие проверяемое исключение **InterruptedException**.

## Пример создания и использования экземпляра TimeUnit

```
package com.gmail.tsa;
```

```
import java.util.concurrent.TimeUnit;
```

```
public class Main {
```

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

```
        TimeUnit tu = TimeUnit.SECONDS;
```

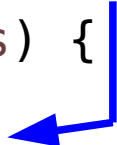
```
        long t = tu.convert(1, TimeUnit.HOURS);
```

```
        System.out.println(t);
```


```
    }
```

```
}
```

Создание. Единицами хранения будут секунды.



Перевод 1 часа в секунды. Так как tu инициализирована секундами (указанно как параметр при создании), то любой перевод будет в секунды.



## Пример использования исполнителя работающего с планировщиком

Постановка задачи: Написать класс, который будет в параллельном потоке выводить текущее время на экран. Вызов этого потока будет осуществляться каждые 3 секунды, пауза между стартом потока 10 секунд.


```
package com.gmail.tsa;

import java.text.SimpleDateFormat;

public class TimeViewer implements Runnable {

    @Override
    public void run() {
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss");
        System.out.println(sdf.format(System.currentTimeMillis()));
    }

}
```



Переопределение run(). То, что в нем описано, будет выполнено в параллельном потоке.


Runnable задача. Ее суть заключается в разовом выводе на экран текущего времени.

# Пример использования исполнителя работающего с планировщиком

```
package com.gmail.tsa;  
  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.ScheduledFuture;  
import java.util.concurrent.TimeUnit;
```

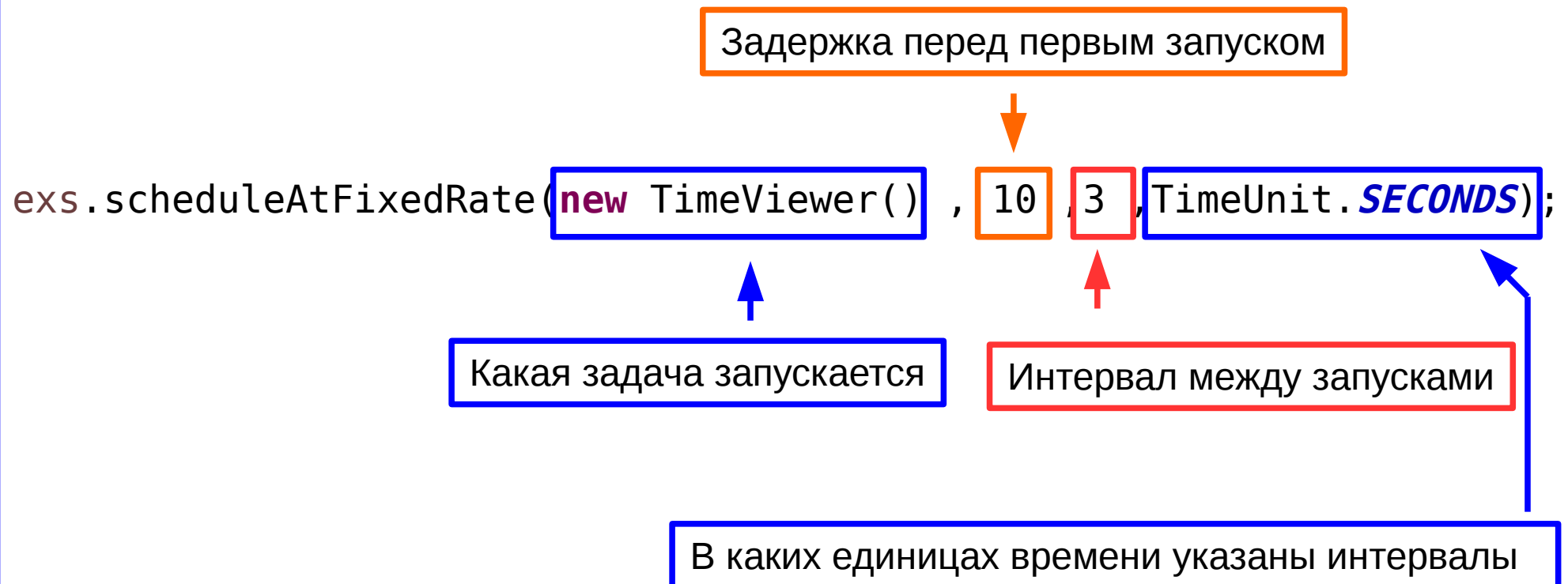
```
public class Main {  
  
    public static void main(String[] args) {  
  
        startTimeViewer();  
  
        System.out.println("END");  
    }  
  
    public static void startTimeViewer() {  
  
        ScheduledExecutorService exs = Executors.newSingleThreadScheduledExecutor();  
  
        ScheduledFuture<?> res = exs.scheduleAtFixedRate(new TimeViewer(), 10, 3,  
            TimeUnit.SECONDS);  
  
    }  
}
```

Создание одного потока исполнителя с планировщиком



Запуск на циклическое выполнение этой задачи.

## Подробнее о запуске задачи с планировщиком



## Итоги урока

Существуют удобные средства для запуска и обработки потоков. К ним относятся исполнители и интерфейсы, позволяющие получать решения, полученные при выполнении задач в параллельных потоках.

В иерархии исполнителей центральное место принадлежит:

- **Executor**
- **ExecutorServices**
- **ScheduledExecutorService**

В большинстве случаев для получения экземпляров класса, реализующего заданные интерфейсы, используются статические методы класса **Executors**.

Полученные реализации позволяют организовать потоковый пул, что существенно упрощает построение многопоточного приложения. Также существует возможность использования потокового пула с планировщиком, что позволяет управлять процессами запуска потоков во времени.

Интерфейс **Callable<T>** позволяет реализовать потоки, которые возвращают результат своей работы. Эти результаты могут быть использованы с помощью интерфейса **Future<T>**.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 1028-1035
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала Java . Том 1. Основы. 9 издание. стр. 816-826



# Java OOP

## «Многопоточное программирование. Часть 2»

Синхронизация потоков — приведение двух или нескольких потоков к такому их протеканию, когда определенные стадии разных потоков совершаются в определенном порядке, либо одновременно.

Синхронизация необходима в любых случаях, когда параллельно протекающим потокам необходимо взаимодействовать. Для ее организации используются средства межпоточкового взаимодействия.

## О необходимости синхронизации к объекту

При работе в многопоточном режиме часто встает задача синхронизации объекта. Т.е. если к одному и тому же объекту одновременно обратятся несколько потоков, то одновременно работать с ним может только один.

Например, предположим есть класс — счет. Его параметрами являются количество денег на счету, логин и пароль пользователя, который этим счетом владеет. Скорее всего объект такого класса будет использоваться в серверной программе (например, клиент — Банк) и работа с ним будет проводиться в многопоточном режиме.

Рассмотрим процедуру снятия наличности со счета:

- 1) Проверка корректности логина и пароля;
- 2) Проверка того что запрашиваемая сумма меньше суммы на счете;
- 3) Выполнение транзакции (обычно самая длинная операция);
- 4) Изменение баланса;
- 5) Печать чека;

Далее симитируем работу в многопоточном режиме и отдадим двум потокам ссылку на один и тот же объект.

```
package com.gmai.tsa;
```

## Класс Account (счет)

```
public class Account {  
    private int money;  
    private String login;  
    private long password;  
    public Account(int money, String login, long password) {  
        super();  
        this.money = money;  
        this.login = login;  
        this.password = password;  
    }  
    public void takeMoney(String login, long password, int sum) {  
        if (!checkPassAndLogin(login, password)) {  
            System.out.println("Wrong login or password");  
            return;  
        }  
        if (!checkMoney(sum)) {  
            System.out.println("You don't have money");  
            return;  
        }  
        transaction();  
        changeBalance(sum);  
        System.out.println(this);  
    }  
}
```

```
private boolean checkPassAndLogin(String login, long password) {  
    return login.equals(this.login) && this.password == password;  
}  
private boolean checkMoney(int money) {  
    return this.money >= money;  
}  
private void transaction() {  
    try {  
        Thread.sleep(500);  
    } catch (InterruptedException e) {  
        System.out.println(e);  
    }  
}  
private void changeBalance(int money) {  
    this.money -= money;  
}
```

```
@Override  
public String toString() {  
    return "Account [money=" + money + "];"  
}  
}
```

Методы проверок и снятия денег.  
Метод transaction() специально  
задерживает основной поток,  
имитируя долгую работу.



## Класс описывающий поток

```
package com.gmai.tsa;
```

```
public class SingleThread implements Runnable {
```

```
    private Account account;
```

```
    private String login;
```

```
    private long password;
```

```
    private int sum;
```



Ссылка на объект класса Account

```
    public SingleThread(Account account) {
```

```
        super();
```

```
        this.account = account;
```

```
    }
```

```
    public void getMoneyFromAccount(String login, long password, int sum) {
```

```
        this.password = password;
```

```
        this.login = login;
```

```
        this.sum = sum;
```

```
        Thread threat = new Thread(this);
```

```
        threat.start();
```

```
    }
```

```
    public void run() {
```

```
        account.takeMoney(this.login, this.password, this.sum);
```

```
    }
```

```
}
```



В параллельном потоке происходит процедура снятия денег для объекта Account, полученного по ссылке.

## Главный метод программы

```
package com.gmai.tsa;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        // TODO Auto-generated method stub
```

Создание объекта класса Account

```
        Account account = new Account(200, "login", 123);
```

```
        SingleThread stOne = new SingleThread(account);  
        SingleThread stTwo = new SingleThread(account);
```

```
        stOne.getMoneyFromAccount("login", 123, 150);  
        stTwo.getMoneyFromAccount("login", 123, 150);
```

```
    }
```

```
}
```

Старт обоих потоков

Двум потокам по ссылке передается один и тот же объект

```
Account [money=-100]  
Account [money=-100]
```

Вывод на экран. Как вы видите, это явно ошибочный вывод, так как второй поток не должен был снять денег.

## Причины некорректной работы приведенного примера

Сначала оба потока одновременно прошли проверки на корректность логина и пароля. Так как ни один из них еще не провел транзакцию то проверку на достаточность суммы также прошли оба. Потом оба прошли транзакцию, и выполнили снятие денег, что и привело к такому результату.

## Возможные пути исправления

Описать логику, согласно которой, если один поток (например 1) начинает выполнение метода этого объекта, то при попытке доступа 2-го потока он бы ожидал окончание работы 1-го и только потом мог вызвать этот метод. Это и есть **синхронизация** доступа к объекту.

## Синхронизация

Для корректной работы необходимо, чтобы одновременно к ресурсу мог только один поток. Процесс обеспечения этого называется **синхронизацией**.

Основным элементом синхронизации является монитор — Объект, который используется как взаимоисключающая блокировка — мьютекс. Только один поток может владеть монитором по умолчанию. В Java все объекты имеют неявный монитор.

В Java два пути синхронизации метода:

- 1) Перед определением метода используется ключевое слово **synchronized**;
- 2) Или заключить исполняемый участок кода в блок оператора **synchronized(объект) {**  
Оператор....  
**};**

# Использование synchronized методов

Если перед методом используется ключевое слово **synchronized**, то в метод может входить только один поток, и до завершения потока другие потоки не могут обратиться к этому методу.

Общий вид объявления

Доступ **synchronized** тип\_возвращаемого\_значения имя\_метода(параметры)

Например

Модификатор доступа

Тип возвращаемого значения

**public synchronized void** takeMoney(String login, **long** password, **int** sum)

Метод синхронизирован

Имя метода

Список параметров



## О работе синхронизированного метода

### Внимание:

- 1) При вызове синхронизированного метода происходит блокирование всего синхронизированного содержимого объекта для доступа из других потоков. Т.е. все синхронизированные методы этого объекта становятся заблокированными для других потоков до момента окончания работы вызванного метода.
- 2) Синхронизация в Java гарантирует, что никакие два потока не смогут выполнить синхронизированный метод одновременно или параллельно.
- 3) Если сделать синхронизированным статический метод, то синхронизация будет выполнена по классу, т. е., будет заблокирован этот метод для всех объектов данного класса.
- 4) Вполне возможно, что и статический и не статический синхронизированные методы могут работать одновременно или параллельно, потому что они захватывают замок на другой объект.
- 5) В соответствии со спецификацией языка вы не можете использовать **synchronized** в конструкторе. Это приведет к ошибке компиляции.

```
package com.gmai.tsa;
```

```
public class Account {  
    private int money;  
    private String login;  
    private long password;  
    public Account(int money, String login, long password) {  
        super();  
        this.money = money;  
        this.login = login;  
        this.password = password;  
    }
```

## Класс Account (счет) с синхронизацией

Теперь метод синхронизирован

```
    public synchronized void takeMoney(String login, long password, int sum) {  
        if (!checkPassAndLogin(login, password)) {  
            System.out.println("Wrong login or password");  
            return;  
        }  
        if (!checkMoney(sum)) {  
            System.out.println("You don't have money");  
            return;  
        }  
        transaction();  
        changeBalance(sum);  
        System.out.println(this);  
    }
```

```
    private boolean checkPassAndLogin(String login, long password) {  
        return (login.equals(this.login) && this.password == password);  
    }  
    private boolean checkMoney(int money) {  
        return this.money >= money;  
    }  
    private void transaction() {  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            System.out.println(e);  
        }  
    }  
    private void changeBalance(int money) {  
        this.money -= money;  
    }  
    @Override  
    public String toString() {  
        return "Account [money=" + money + "];"  
    }  
}
```

Методы проверок и снятия денег. Метод transaction() специально задерживает основной поток, имитируя долгую работу.

## Результат работы синхронизированного метода

Вывод на экран:

```
Account [money=50]  
You don't have money
```

Теперь можно видеть, что программа отработала корректно. Сначала один из потоков захватил блокировку объекта, тем самым переводя второй поток в режим ожидания. Как только первый поток отработал, он освободил блокировку, что привело к активизации второго потока. Однако он уже не прошел проверку на сумму, что и ожидалось от этой программы.

## Использование оператора `synchronized`

Если у вас нет возможности написать синхронизированный метод, либо по каким-то причинам, у вас нет доступа к исходному коду метода тогда используется оператор **`synchronized`**.

Общий вид объявления

```
synchronized (объект) {  
    Операторы....  
}
```

Объект — ссылка на объект, методы которого вы хотите синхронизировать.

! Если вы вызовете эти методы вне оператора **`synchronized`**, то они вызовутся как не синхронизированные.

```
package com.gmai.tsa;
```

```
public class Account {  
    private int money;  
    private String login;  
    private long password;  
    public Account(int money, String login, long password) {  
        super();  
        this.money = money;  
        this.login = login;  
        this.password = password;  
    }  
    public void takeMoney(String login, long password, int sum) {
```

```
        synchronized (this) {  
            if (!checkPassAndLogin(login, password)) {  
                System.out.println("Wrong login or password");  
                return;  
            }  
            if (!checkMoney(sum)) {  
                System.out.println("You don't have money");  
                return;  
            }  
            transaction();  
            changeBalance(sum);  
            System.out.println(this);  
        }  
    }
```

```
private boolean checkPassAndLogin(String login, long password) {  
    return (login.equals(this.login) && this.password == password);  
}  
private boolean checkMoney(int money) {  
    return this.money >= money;  
}  
private void transaction() {  
    try {  
        Thread.sleep(500);  
    } catch (InterruptedException e) {  
        System.out.println(e);  
    }  
}  
private void changeBalance(int money) {  
    this.money -= money;  
}  
@Override  
public String toString() {  
    return "Account [money=" + money + "];"  
}  
}
```

## Класс Account (счет) использующий synchronized

Использование оператора  
**synchronized**



Часто используемый вариант синхронизации,  
когда синхронизируется текущий объект  
(объектом выступает **this**).

## Какой подход и когда использовать?

В общем случае результат работы обоих подходов будет одинаковым.

Если вы пишете приложение, которое точно будет работать в многопоточном режиме, тогда лучше писать синхронизированные методы.

Если приложение написано не вами, и нельзя модифицировать исходный код, либо при написании приложение оно не было рассчитано на многопоточное выполнение, тогда следует использовать оператор `synchronized`.

Таким образом базовым механизмом для синхронизации объекта в многопоточном режиме работы является либо использование **synchronized** методов, либо использование оператора **synchronized**.

Однако данный подход **не может решить** задачу, когда поток нужно принудительно приостановить на объекте и активировать другой, в зависимости от какого-то «события». Также часто решается задача об активации остановленного потока для его дальнейшего выполнения. Для этого используется подход с использованием межпоточкового взаимодействия и синхронизация по событию.

## Межпоточковые коммуникации

Применяются для синхронизации на уровне событий, а не на уровне данных. Т.е мы хотим, чтобы поток приостанавливался или стартовал при наступлении требуемого события.

Так как поток также является объектом, то для него справедливы методы суперкласса `Object`.

**`Object.wait()`** – заставляет текущий поток ждать пока другой поток не вызовет метод `notify()/notifyAll()` для данного объекта.

**`Object.notify()`** – пробуждает один поток, который ждет события, и предварительно вызвал **`wait()`**.

**`Object.notifyAll()`** – пробуждает все потоки, которые ждут события, и предварительно вызвали **`wait()`**.

**!Все эти методы могут быть вызваны только из синхронизированного контекста.**

# Наиболее простая форма межпоточковых коммуникаций

Создать объект-посредник.

## Action

int value

boolean turn

synchronized void setValue (int value)

synchronized int getValue ()

## ThreadTwo

Action action

int value

## ThreadOne

Action action

int value

Создать два потока и передать им ссылку на один объект-посредник.

Т.е. обмен осуществляется через класс посредник. Один поток, вызывая метод по установке **setValue (int value)**, устанавливает свойство класса (при этом изменяя значение переменной turn, чтобы сигнализировать о новом изменении). Второй поток, вызывая метод получения **getValue()**, получит значение свойства. Т.е. по сути произойдет передача данных из потока 1 в поток 2.



## Пример реализации межпотокowego сообщения

Создадим два потока:

1-й поток будет генерировать случайное целое число и отправлять его, 2-й поток будет получать это число.

Если бы потоки не были синхронизированы, то процесс отправки-получения был бы хаотичным, т. е. могло быть отправлено несколько значений подряд, а принято только последнее. Или наоборот несколько раз принято одно и то же значение.

Реализовать возможность корректной работы этой программы, используя синхронизацию по объекту, крайне сложно. Намного более простым способом является использование синхронизации по событию. Для этого построим проект на основе описанных классов потоков и класса посредника.

# Класс-посредник для потоков

```
package com.gmail.tsa;
```

```
public class Action {
```

```
    private int value;
```

Свойство для передачи данных от потока 1 к потоку 2

```
    private boolean turn = false;
```

Свойство для указания очередности работы потоков

```
    private boolean stop = false;
```

Свойство для указания потоку 2 завершить свою работу

```
    public Action() {  
        super();  
    }
```

```
    public synchronized int getValue() {
```

Метод для отдачи значения.

```
        for (; turn == false;) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }
```

Обратите внимание, что блокирование потока происходит в цикле. Это связано с возможностью самопроизвольного старта заблокированного потока

```
        int temp = this.value;  
        turn = false;  
        notifyAll();  
        System.out.println("Number accepted -> " + this.value);  
        return temp;  
    }
```

Активация заблокированных до этого потоков

```
    public synchronized void setValue(int value) {
```

Метод для получения значения.

```
        for (; turn == true;) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }
```

Обратите внимание, что в этом цикле условие блокировки потока противоположно предыдущему. Это гарантирует поочередность работы потоков.

```
        this.value = value;  
        turn = true;  
        System.out.println("Send number -> " + this.value);  
        notifyAll();  
    }
```

```
    public boolean isStop() {  
        return stop;  
    }
```

Метод для установки и получения свойства ответственного за окончание работы 2 потока.

```
    public void setStop(boolean stop) {  
        this.stop = stop;  
    }
```

```
}
```

## Класс передающий данные

```
package com.gmail.tsa;
```

```
import java.util.Random;
```

```
public class Provider implements Runnable {
```

```
    private Action action;
```

Одним из свойств класса является ссылка на объект посредник.

```
    private Random rn = new Random();
```

```
    public Provider(Action action) {  
        super();  
        this.action = action;  
    }
```

Ссылка на объект класса-посредника получается в конструкторе.

```
@Override
```

```
public void run() {  
    for (int i = 0; i < 10; i++) {  
        int randomNumber = rn.nextInt(100);
```

Используем синхронизированный метод для отправки данных.

```
        action.setValue(randomNumber);
```

```
    }
```

```
    action.setStop(true);
```

Устанавливаем свойство сигнализирующее о том что вычисления закончились.

```
}
```

```
}
```

Так как реализуется Runnable, то на основе этого класса можно создать поток.

## Класс принимающий данные

```
package com.gmail.tsa;
```

Так как реализуется Runnable, то на основе этого класса можно создать поток.

```
public class Receiver implements Runnable {
```

```
    private Action action;
```

Одним из свойств класса является ссылка на объект посредник.

```
    public Receiver(Action action) {  
        super();  
        this.action = action;  
    }
```

Ссылка на объект класса-посредника получается в конструкторе.

```
@Override
```

```
    public void run() {  
        for (; !action.isStop();) {
```

Обратите внимание, что поток работает до тех пор, пока свойство stop класса посредника равно false.

```
            int number = action.getValue();
```

Используем синхронизированный метод для получения данных.

```
        }
```

```
    }
```

```
}
```

# Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Action action = new Action();
```

Создание экземпляра класса-посредника

```
        Provider provider = new Provider(action);
```

```
        Receiver receiver = new Receiver(action);
```

```
        Thread threadOne = new Thread(provider);
```

```
        Thread threadTwo = new Thread(receiver);
```

```
        threadOne.start();
```

```
        threadTwo.start();
```

```
    }
```

```
}
```

Создание и запуск потоков на основе описанных выше классов

Создание классов для передачи и для приема данных. Обратите внимание, что им, в качестве параметра конструктора, передается ссылка на один и тот же объект класса-посредника.

## Вывод на консоль результатов работы программы

```
Send number -> 57  
Number accepted -> 57  
Send number -> 14  
Number accepted -> 14  
Send number -> 74  
Number accepted -> 74  
Send number -> 3  
Number accepted -> 3  
Send number -> 82  
Number accepted -> 82  
Send number -> 22  
Number accepted -> 22  
Send number -> 12  
Number accepted -> 12  
Send number -> 98  
Number accepted -> 98  
Send number -> 68  
Number accepted -> 68  
Send number -> 25  
Number accepted -> 25
```

Видно что, потоки работают поочередно. Без утери или дублирования данных.

## Взаимная блокировка(deadlock) в Java

Взаимная блокировка – это ситуация, в которой два или более процесса, занимая некоторые ресурсы, пытаются заполучить другие ресурсы, занятые другими процессами, и ни один из процессов не может занять необходимый им ресурс, и соответственно, освободить занимаемый.

Ниже будет представлен пример, когда один объект вызывает из синхронизированного метода синхронизированный метод другого объекта, который, в свою очередь, обратиться к другому синхронизированному методу первого объекта. Далее вызовем метод первого объекта из параллельного потока, тем самым активировав его монитор. В результате поток, связанный со вторым объектом, будет вечно ждать окончания первого потока и наоборот. Т. е. произойдет взаимная блокировка.

## Пример взаимной блокировки в Java

```
package com.gmail.tsa;

public class A {
    private int value;
    private B classB;

    public A(int value) {
        super();
        this.value = value;
    }

    public void setClassB(B classB) {
        this.classB = classB;
    }

    public synchronized int getValue() {
        System.out.println(Thread.currentThread().getName());
        System.out.println("Method getValue() class A");
        return this.value;
    }

    public synchronized int getSumm() {
        System.out.println(Thread.currentThread().getName());
        System.out.println("Method getSumm() class A");
        System.out.println("Class A call method getValue() class B");
        return classB.getValue() + this.value;
    }
}
```



## Пример взаимной блокировки в Java

```
package com.gmail.tsa;

public class B {
    private int value;
    private A classA;

    public B(int value) {
        super();
        this.value = value;
    }

    public void setClassA(A classA) {
        this.classA = classA;
    }

    public synchronized int getValue() {
        System.out.println(Thread.currentThread().getName());
        System.out.println("Method getValue() class B");
        System.out.println("Class B call method getValue() class A");
        return classA.getValue() + this.value;
    }
}
```

Как можно видеть, классы A и B взаимно вызывают синхронизированные методы друг друга. Если хоть один из них уже окажется заблокирован то это гарантированно приведет к взаимной блокировке.

## Пример взаимной блокировки в Java

```
package com.gmail.tsa;

public class SingleThread implements Runnable {
    private B classB;

    public void setClassB(B classB) {
        this.classB = classB;
    }

    @Override
    public void run() {
        System.out.println(classB.getValue());
    }
}
```

Этот поток также попытается захватить объект класса В. А так как объект класса А будет заблокирован главным потоком, то это может привести к взаимной блокировке.

## Пример взаимной блокировки в Java

```
package com.gmail.tsa;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        A classA = new A(10);
        B classB = new B(12);

        classA.setClassB(classB);
        classB.setClassA(classA);
        SingleThread sthread = new SingleThread();
        sthread.setClassB(classB);
        Thread cuThread = new Thread(sthread);
        cuThread.start();
        System.out.println(classA.getSumm());
    }
}
```

В результате блокировки главным потоком объекта класса А блокируется поток, который хочет вызвать его синхронизированный метод. Однако параллельный поток блокирует объект класса В. А так как главный поток через объект класса А пытается вызвать его метод, это приводит к взаимной блокировке.

# О проблемах многопоточного прямого доступа к свойству класса

Предположим, есть класс с открытым свойством. Это свойство попытаются изменить одновременно несколько потоков. При этом они не только его изменяют, а и используют для вывода на экран. Так как эта переменная не синхронизирована по доступу, то могут встречаться дубли и пропуски.

```
package com.gmail.tsa;
```

```
public class SomeClass {  
    public Integer volume = 0;  
}
```

Открытое свойство класса, для изменения в многопоточной среде

```
package com.gmail.tsa;
```

```
public class SingleThread implements Runnable {  
    private SomeClass someClass;  
  
    public SingleThread(SomeClass someClass) {  
        super();  
        this.someClass = someClass;  
    }  
  
    @Override  
    public void run() {  
        int x = someClass.volume ++;  
        System.out.print(" " + x);  
    }  
}
```

В параллельном потоке сначала изменяется значение, а потом выводится на экран.

# О проблемах многопоточного прямого доступа к свойству класса

Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        SomeClass smClass = new SomeClass();
```

Создание одного объекта

```
        for (int i = 0; i < 10; i++) {  
            Thread thread = new Thread(new SingleThread(smClass));  
            thread.start();  
        }
```

Его использование в 10 потоках

Один из возможных выводов в консоль

2 2 3 4 5 6 7 8 10 10

Как вы видите, есть много дублей (в примере 2). Это связано с тем, что потоки работают с копиями переменных, которые «не успевают» синхронизироваться между операцией изменения и вывода на экран.

## Немного о ключевом слове **volatile**

Определение переменной с ключевым словом **volatile** означает, что значение этой переменной может изменяться другими потоками. И это дает автоматическую синхронизацию для атомарных операторов над этой переменной.

В целях повышения производительности спецификация языка Java допускает сохранение в JRE локальной копии переменной для каждого потока, который на нее ссылается. Такие "локальные" копии переменных напоминают кэш и помогают потоку избежать обращения к главной памяти каждый раз, когда требуется получить значение переменной. При запуске двух потоков один из них считывает переменную A как 5, а второй — как 10. Если значение переменной A изменилось с 5 на 10, то первый поток не узнает об изменении и будет хранить неправильное значение A. Но если переменная A помечена как **volatile**, то когда бы поток не считывал значение A, он будет обращаться к главной копии A и считывать ее текущее значение. Локальный кэш потока имеет смысл в том случае, если переменные в ваших приложениях не будут изменяться извне.

Если переменная объявлена как **volatile**, это означает, что она может изменяться разными потоками. Естественно ожидать, что JRE обеспечит ту или иную форму синхронизации таких volatile-переменных. JRE действительно неявно обеспечивает синхронизацию при доступе к volatile-переменным, но с одной очень большой оговоркой: чтение volatile-переменной и запись в volatile-переменную синхронизированы, а **неатомарные операции — нет**.

## Новые средства для работы с многопоточностью

В JDK 7, а в последствии 1.8, значительно расширился набор средств для работы с многопоточностью. Они являются частью пакета **java.util.concurrent**.

В этом классе описаны объекты типа AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray.

AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray позволяют использовать многопоточный прямой доступ к свойству класса без использования блокировки.

## Пример применения AtomicInteger

Пример синхронизации прямого доступа в многопоточном режиме с использованием Atomic переменных. Стоит обратить внимание на то, что синхронизированы даже не атомарные операции (в данном примере инкремент).

```
package com.gmail.tsa;

import java.util.concurrent.atomic.AtomicInteger;

public class SomeClass {
    public AtomicInteger volume = new AtomicInteger(0);
}
```

Открытое свойство типа Atomic




```
package com.gmail.tsa;

public class SingleThread implements Runnable {
    private SomeClass someClass;

    public SingleThread(SomeClass someClass) {
        super();
        this.someClass = someClass;
    }

    @Override
    public void run() {
        int x = someClass.volume.getAndIncrement();
        System.out.print(" " + x);
    }
}
```

В параллельном потоке  
сначала изменяется значение,  
а потом выводится на экран.





# Пример использования Atomic переменной

Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        SomeClass smClass = new SomeClass();
```

Создание одного объекта

```
        for (int i = 0; i < 10; i++) {  
            Thread thread = new Thread(new SingleThread(smClass));  
            thread.start();  
        }
```

Его использование в 10 потоках

Один из возможных выводов в консоль

0 1 2 3 4 5 6 7 8 9

В случае использования Atomic переменных синхронизация прямого доступа к свойству класса выполняется автоматически. Это может существенно уменьшить размер кода.

## Итоги урока

Синхронизация потоков — приведение двух или нескольких потоков к такому их протеканию, когда определенные стадии разных потоков совершаются в определенном порядке, либо одновременно.

Синхронизация чаще всего выполняется:

- Для предоставления очередности использования потоками объекта (синхронизация по объекту) - используются **synchronized** методы или оператор **synchronized()**
- Для синхронизации по какому-нибудь событию — используются методы **wait()**, **notify()**, **notifyAll()**
- Для синхронизации прямого доступа к свойству класса проще всего использовать **Atomic** переменные.

При неправильном планировании многопоточного приложения может возникнуть взаимная блокировка потоков, что приведет к неработоспособности вашего приложения.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 300 — 315
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала Java . Том 1. Основы. Девятое издание. стр. 774-800

## Домашнее задание

- 1) Существуют три корабля. На каждом из них 10 ящиков груза. Они одновременно прибыли в порт, в котором только два дока. Скорость разгрузки - 1 ящик в 0.5 сек. Напишите программу, которая, управляя кораблями, позволит им правильно разгрузить груз.
- 2) Реализуйте программу многопоточного копирования файла блоками с выводом прогресса на экран.
- 3) Реализуйте процесс многопоточного поиска файла в файловой системе. Т.е. вы вводите название файла и в какой части файловой системы его искать. Программа должна вывести на экран все адреса файлов с таким названием.

# Синхронизаторы

## Дополнительный материал к лекции

### «Многопоточное программирование. Часть 2 »

Синхронизаторы — объекты, блокирующие или активирующие заблокированные ранее потоки выполнения в зависимости от заданных условий.

Составил: Цымбалюк А.Н.

# Java Lock API

`java.util.concurrent.locks` — пакет, в котором описаны дополнительные синхронизаторы. Основу составляет интерфейс `Lock` и некоторые дополнительные классы, которые усовершенствовали механизм блокировки.

Интерфейс `Lock` описывает базовые механизмы синхронизации

Метод	Описание
<code>void lock()</code>	Устанавливает блокировку потока.
<code>void lockInterruptibly()</code>	Устанавливает блокировку, если текущий поток не прерван. Также есть возможность прервать ожидающий поток.
<code>Condition newCondition()</code>	Возвращает объект типа <code>Condition</code> — условие управляющее блокировкой.
<code>boolean tryLock()</code>	Попытка установки блокировки. Если блокировку установить не удалось, вернет <code>false</code> .
<code>boolean tryLock(long time, TimeUnit unit)</code>	Попытка получить блокировку. Если за <code>time</code> единиц времени (указывается с помощью <code>unit</code> ) попытка неудачна, тогда отказ.
<code>void unlock()</code>	Снимает блокировку.

## Реализация интерфейса Lock

Пакет `java.util.concurrent.locks` включает три реализации интерфейса `Lock`:

- `ReentrantLock`
- `ReentrantReadWriteLock.ReadLock`
- `ReentrantReadWriteLock.WriteLock`

Наиболее часто используемая реализация — `ReentrantLock`. Средство для установки реентабельной блокировки. Реентабельная блокировка - блокировка, которая может быть повторно захвачена потоком владельцем. Эта блокировка снабжена счетчиком захватов. Благодаря этому средству код, защищенный блокировкой, может вызвать другой метод, использующий ту же самую блокировку.

## Пример синхронизации с использованием ReentrantLock

Для сравнения используем задачу из базового курса (Многопоточное программирование. Часть 2). Ее условие приведено ниже:

Например : предположим есть класс — счет. Его параметрами являются количество денег на счету, логин и пароль пользователя который этим счетом владеет. Скорее всего объект такого класса будет использоваться в серверной программе (например клиент — Банк) и работа с ним будет проходить в многопоточном режиме.

Рассмотрим процедуру снятия наличности со счета:

- 1) Проверка корректности логина и пароля;
- 2) Проверка того, что запрашиваемая сумма меньше суммы на счете;
- 3) Выполнение транзакции (обычно самая длинная операция);
- 4) Изменение баланса;
- 5) Печать чека;

Далее симитируем работу в многопоточном режиме и отдадим двум потокам ссылку на один и тот же объект.



## Класс Account (счет)

```
package com.gmail.tsa;  
  
import java.util.concurrent.locks.ReentrantLock;
```

```
public class Account {  
    private int money;  
    private String login;  
    private long password;
```

```
    private ReentrantLock lock = new ReentrantLock();
```

Объект класса ReentrantLock для синхронизации

```
    public Account(int money, String login, long password) {  
        super();  
        this.money = money;  
        this.login = login;  
        this.password = password;  
    }
```

```
    public void takeMoney(String login, long password, int sum) {  
        try {  
            lock.lock();  
            if (!checkPassAndLogin(login, password)) {  
                System.out.println("Wrong login or password");  
                return;  
            }  
            if (!checkMoney(sum)) {  
                System.out.println("You don't have money");  
                return;  
            }  
            transaction();  
            changeBalance(sum);  
            System.out.println(this);  
        }
```

Блокировка потока для синхронизации

```
        finally {  
            lock.unlock();  
        }
```

Разблокировка потока. Обратите внимание на то, что он выполняется в блоке finally. Т.е. сгенерируется исключение или нет, поток все равно разблокируется.

```
    private boolean checkPassAndLogin(String login, long password) {  
        return (login.equals(this.login) && this.password == password);  
    }  
    private boolean checkMoney(int money) {  
        return this.money >= money;  
    }  
    private void transaction() {  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            System.out.println(e);  
        }  
    }  
    private void changeBalance(int money) {  
        this.money -= money;  
    }
```

Методы проверок и снятия денег. Метод transaction() специально задерживает основной поток, имитируя долгую работу.

```
    @Override  
    public String toString() {  
        return "Account [money=" + money + " ]";  
    }  
}
```

```
package com.gmai.tsa;
```

## Класс описывающий поток.

```
public class SingleThread implements Runnable {
```

```
    private Account account;
```

```
    private String login;
```

```
    private long password;
```

```
    private int sum;
```



Ссылка на объект класса Account

```
    public SingleThread(Account account) {
```

```
        super();
```

```
        this.account = account;
```

```
    }
```

```
    public void getMoneyFromAccount(String login, long password, int sum) {
```

```
        this.password = password;
```

```
        this.login = login;
```

```
        this.sum = sum;
```

```
        Thread threat = new Thread(this);
```

```
        threat.start();
```

```
    }
```

```
    public void run() {
```

```
        account.takeMoney(this.login, this.password, this.sum);
```

```
    }
```

```
}
```



В параллельном потоке происходит процедура снятия денег для объекта Account полученного по ссылке.

# Главный метод программы

```
package com.gmai.tsa;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        // TODO Auto-generated method stub
```

Создание объекта класса Account

```
        Account account = new Account(200, "login", 123);
```

```
        SingleThread stOne = new SingleThread(account);  
        SingleThread stTwo = new SingleThread(account);
```

```
        stOne.getMoneyFromAccount("login", 123, 150);  
        stTwo.getMoneyFromAccount("login", 123, 150);
```

```
    }
```

```
}
```

Старт обеих потоков

Двум потокам по ссылке передается один и тот же объект

```
Account [money=50]  
You don't have money
```

Вывод на экран. Код синхронизирован по объекту, т. е. выполнение аналогично выполнению **synchronized** метода

## Подробнее о ReentrantLock

В наиболее простом случае объект класса ReentrantLock можно использовать как замену **synchronized** метода. Однако он обладает более широкими возможностями.

Например : метод tryLock() только попытается захватить блокировку и, если это удалось, то этот метод вернет true, а если нет, то поток можно переключить на выполнение иных действий. Т.е. такой метод значительно снизит вероятность взаимной блокировки.

Существует перегруженная версия этого метода, которая позволяет попробовать захватить блокировку в течении указанного времени.

tryLock(time, TimeUnit) — пробовать time единиц времени, выраженных с помощью TimeUnit.

Полезной особенностью метода tryLock() и lockInterruptibly() является возможность прерывания потока находящегося в режиме ожидания.

## Пример использования ReentrantLock.tryLock()

```
package com.gmail.tsa;
```

```
import java.util.concurrent.locks.ReentrantLock;
```

```
public class SynchClass {
```

```
    private long count = 12;
```

```
    private ReentrantLock lock = new ReentrantLock();
```

```
    public long getCount() {
```

```
        if (lock.tryLock()) {
```

```
            try {
```

```
                try {
```

```
                    Thread.sleep(2000);
```

```
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
```

```
                }
```

```
                System.out.println(Thread.currentThread() + " - getValue");
```

```
                return this.count;
```

```
            } finally {
```

```
                lock.unlock();
```

```
            }
```

```
        } else {
```

```
            System.out.println(Thread.currentThread() + " - do something");
```

```
            return -1;
```

```
        }
```

```
    }
```

```
}
```

Если объект еще не заблокирован, то выполнится эта часть

Попытка получить блокировку. Если получилось, работаем. Нет - делаем что-то еще.

Специальное торможение потока для получения эффекта блокировки

Эта часть выполнится, если объект уже заблокирован.

Поток, который попытается захватить объект описанного выше класса

```
package com.gmail.tsa;

public class SingleThread implements Runnable {

    private SynchClass synchClass;

    public SingleThread(SynchClass synchClass) {
        super();
        this.synchClass = synchClass;
    }

    @Override
    public void run() {
        System.out.println(synchClass.getCount());
    }

}
```

Т.е. описанный поток просто попытается вызвать метод (синхронизированный с помощью ReentrantLock.tryLock())

## Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

Один объект получили оба потока

```
        SynchClass synchClass = new SynchClass();
```

```
        SingleThread sThreadOne = new SingleThread(synchClass);  
        SingleThread sThreadTwo = new SingleThread(synchClass);
```

```
        Thread threadOne = new Thread(sThreadOne);  
        Thread threadTwo = new Thread(sThreadTwo);
```

```
        threadOne.start();  
        threadTwo.start();
```

```
    }
```

```
}
```

Создание и запуск потоков

```
Thread[Thread-1,5,main] - do something  
-1  
Thread[Thread-0,5,main] - getValue  
12
```

Вывод в консоль. Как видно, один поток заблокировал поток, второй поток не смог получить блокировку и выполнил другое действие.

## Объекты условий

Объекты условий стоит использовать в случае, когда поток захватил блокировку объекта, но оказалось, что из-за каких-то условий его продолжение не имеет смысла. В таком случае его можно перевести в режим ожидания выполнения данного условия. Для этого используются объекты условий объектов класса **Condition**.

`Java.util.concurrent.Lock.Condition`

Метод	Описание
<code>void await()</code>	Выводит поток в режим ожидания по условию.
<code>void signalAll()</code>	Разблокирует все потоки в режиме ожидания по данному условию.
<code>void signal()</code>	Разблокирует поток в режиме ожидания по данному условию.

Внимание! Как было указано ранее, объект класса **Condition** создается с помощью метода **newCondition()** объекта класса, реализующего интерфейса **Lock**.



## Использование объекта условий для синхронизации

Есть класс — фабрика юнитов. Известно, что построение одного юнита требует определенного количества пси (Да, да! StarCraft - одна из моих любимых игр). К этой фабрике могут одновременно обращаться несколько потоков. Один из них строит юниты, тем самым потребляя пси. Другой поток строит пилоны (источники пси), тем самым добавляя пси.

Задача синхронизации: предположим, поток потребляющий пси, захватил блокировку фабрики, но для построения юнита пси не хватает. Тогда этот поток переводится в режим ожидания для разблокировки объекта. Это позволит другому потоку добавить пси и активировать этот поток.

Такой подход позволит избежать условия взаимной блокировки. Например в наличии 0 пси. При захвате объекта потоком потребителем в обычных условиях он бы навсегда заблокировал этот объект, так как потоку, который бы добавил пси, доступ уже был закрыт.

# Класс Фабрика

```
package com.gmail.tsa;
```

```
import java.util.concurrent.locks.Condition;  
import java.util.concurrent.locks.ReentrantLock;
```

```
public class SynchClass {  
    private long pylons;
```

Количество пси

```
    private ReentrantLock lock = new ReentrantLock();  
    private Condition condition = lock.newCondition();
```

Синхронизатор типа Lock и условие им порожденное

```
    public SynchClass(long pylons) {  
        super();  
        this.pylons = pylons;  
    }
```

```
    public long getPylons(long unitEnergy) {  
        try {
```

Получение блокировки

```
            lock.lock();
```

```
            for (; this.pylons < unitEnergy;) {  
                try {  
                    System.out.println("You Must Construct Additional Pylons!");  
                    condition.await();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }
```

```
            this.pylons -= unitEnergy;  
            System.out.println("Unit return");  
        } finally {  
            lock.unlock();  
        }
```

Проверка условия и если оно не выполнено, блокировка потока по условию

```
        return pylons;  
    }
```

Снятие блокировки

```
    public void setPylons(long pylons) {  
        try {  
            lock.lock();  
            this.pylons = pylons;  
            System.out.println("Constructed " + pylons + " pylons");  
            condition.signalAll();  
        } finally {  
            lock.unlock();  
        }  
    }
```

Метод блокирующий объект при добавлении пси

Активация заблокированных по условию потоков

```
}
```

## Более подробно о использовании объекта условия

```
public long getPylons(long unitEnergy) {  
    try {
```

```
        lock.lock();
```

Поток, вызывающий этот метод, заблокирует объект

```
        for (; this.pylons < unitEnergy;) {
```

Проверяем условие

```
            try {  
                System.out.println("You Must Construct Additional Pylons!");
```

```
                condition.await();
```

Вызывая метод await() объекта условия, блокируем поток по условию

```
            } catch (InterruptedException e) {  
                e.printStackTrace();
```

```
            }
```

```
        }
```

```
        this.pylons -= unitEnergy;
```

```
        System.out.println("Unit return");
```

Обратите внимание, блокировка по условию осуществляется в теле цикла.

```
    } finally {
```

```
        lock.unlock();
```

Разблокировка потока

```
    }
```

```
    return pylons;
```

```
}
```

## Классы потребляющие и генерирующие пси

```
package com.gmail.tsa;

public class BuildPylons implements Runnable {
    private SynchClass synchClass;
    private long pylons;

    public BuildPylons(SynchClass synchClass, long pylons) {
        super();
        this.synchClass = synchClass;
        this.pylons = pylons;
    }

    @Override
    public void run() {
        synchClass.setPylons(pylons);
    }
}
```

```
package com.gmail.tsa;

public class BuildUnit implements Runnable {
    private SynchClass synchClass;
    private long pylons;

    public BuildUnit(SynchClass synchClass, long pylons) {
        super();
        this.synchClass = synchClass;
        this.pylons = pylons;
    }

    @Override
    public void run() {
        synchClass.getPylons(this.pylons);
    }
}
```

## Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

Объект класса фабрики

```
        SynchClass synchClass = new SynchClass(0);
```

```
        BuildUnit bUnit = new BuildUnit(synchClass, 10);
```

```
        BuildPylons bPylons = new BuildPylons(synchClass, 10);
```

```
        Thread threadBuildUnit = new Thread(bUnit);
```

```
        Thread threadBuildPylons = new Thread(bPylons);
```

```
        threadBuildUnit.start();
```

```
        threadBuildPylons.start();
```

Создание и старт потоков

```
    }
```

Двум потокам передается один и тот же объект

```
}
```

## Сравнение использование интерфейса Lock с **synchronized** методами

В общем случае использование объектов реализующих интерфейс Lock дает более расширенный контроль по сравнению с использованием **synchronized** методов. Однако в ряде случаев использование **synchronized** методов достаточно для решения задачи.

В тоже время использование методов wait(), notify(), notifyAll() аналогично использованию объектов условия (**Condition**) с методами await(), signal(), signalAll()

Использование **Lock** оправданно необходимостью более полного контроля за синхронизацией.

## Синхронизаторы в пакете java.util.concurrent

Дополнительный набор инструментов ( зачастую более удобных, чем низкоуровневая синхронизация) содержится в пакете **java.util.concurrent**

Класс	Описание
Semaphore	Доступ к ресурсу на основе счетчика разрешений.
CountDownLatch	Доступ к ресурсу на основе количества произошедших событий.
CyclicBarrier	Позволяет группе потоков исполнения войти в режим ожидания в предварительно заданной точке выполнения.
Exchanger	Обмен данными между двумя потоками исполнения

Следует иметь в виду, что каждый синхронизатор предоставляет конкретное решение задачи синхронизации. Благодаря этому можно оптимизировать работу каждого синхронизатора.

## Класс Semaphore

Реализует синхронизатор типа семафор. Семафор управляет доступом к ресурсу с помощью счетчика. Если значение счетчика больше 0, то доступ разрешается, если меньше или равен 0, то доступ запрещается.

Т.е. при попытке получения доступа к ресурсу, поток запрашивает разрешение у семафора, и если он его получает, выполняется. При этом счетчик семафора уменьшается на нужное количество разрешений. Если поток отработал и доступа к ресурсу ему больше не нужно, то он освобождает определенное количество разрешений, что даст возможность выполниться другим потокам.

Метод	Описание
<code>void acquire()</code>	Получить одно разрешение.
<code>void acquire(int number)</code>	Получить number разрешений.
<code>void release()</code>	Освободить разрешение.
<code>void release(int number)</code>	Освободить number разрешений.

Конструкторы :

- `Semaphore (int number)` — создает семафор с number разрешениями
- `Semaphore (int number, boolean flag)` — создает семафор с number разрешениями, если `flag == true`, то разрешения будут выдаваться потокам в порядке их запроса.



## Пример использования семафора

```
package com.gmail.tsa;  
  
import java.util.concurrent.Semaphore;
```

```
public class SourceClass {  
    private int number;
```

Семафор на пять разрешений



```
    private Semaphore semaphore = new Semaphore(5, true);
```

```
    public SourceClass(int number) {  
        super();  
        this.number = number;  
    }
```

```
    public int getNumber() {  
        try {
```

```
            semaphore.acquire();
```

Получение одного разрешения у семафора



```
            System.out.println(Thread.currentThread().getName() + "work");  
            Thread.sleep(3000);  
        } catch (InterruptedException e) {  
            System.out.println(e);  
        }
```

```
        semaphore.release();
```

Возврат разрешения семафору



```
        return number;  
    }
```

Таким образом данным объектом одновременно могут владеть только 5 потоков.


## Поток, который получает доступ к синхронизируемому объекту

```
package com.gmail.tsa;
```

```
public class SingleThread implements Runnable {
```


```
    private SourceClass sourceClass;
```

Свойство в виде ссылки на синхронизируемый объект




```
    public SingleThread(SourceClass sourceClass) {  
        super();  
        this.sourceClass = sourceClass;  
    }
```

Получение ссылки на объект в конструкторе



```
    @Override  
    public void run() {  
        sourceClass.getNumber();  
    }
```

В параллельном потоке вызовется синхронизированный метод объекта.



```
}
```

## Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

Создание объекта синхронизируемого класса

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

```
        SourceClass sourceClass = new SourceClass(10);
```

```
        for (int i = 0; i < 10; i++) {
```

```
            SingleThread singleThread = new SingleThread(sourceClass);
```

```
            Thread thread = new Thread(singleThread);
```

```
            thread.start();
```

```
        }
```

```
    }
```

```
}
```

Создание 10 потоков и передача им одного и того же объекта по ссылке. Так как объект синхронизирован семафором, то одновременно им может владеть не более 5. Поэтому, сначала выполняться первые 5 потоков, потом (через определенную паузу - она возникла в результате вызова метода `Thread.sleep(3000)`) отработают остальные.

## Класс CountdownLatch

Используется для перевода потока в режим ожидания до тех пор, пока не наступит определенное количество событий.

Метод	Пример
<code>void await()</code>	Ожидание по указанной блокировке.
<code>boolean await(long time, TimeUnit unit)</code>	Ожидает time единиц времени выраженных unit параметром. Вернет true, если обратный отчет разрешений дошел до 0. И false, если закончилось время.
<code>void countDown()</code>	Добавление одного события. И как следствие уменьшение счетчика необходимых событий на 1.
<code>long getCount()</code>	Вернет текущее значение счетчика

### Конструктор:

- `CountDownLatch (int count)` — Создает объект блокировки с count, числом событий, которые должны наступить для разблокировки.

```
package com.gmail.tsa;
```

## Пример использования CountdownLatch

```
import java.util.concurrent.CountDownLatch;
```

```
public class SynchClass {
```

Блокировка будет установлена до 5 событий

```
private CountDownLatch cdLatch = new CountDownLatch(5);
```

```
private int number = 0;
```

```
public SynchClass() {
```

```
    super();
```

```
}
```

```
public int getNumber() {
```

```
    try {
```

```
        System.out.println("I am waiting");
```

```
        cdLatch.await();
```

```
    } catch (InterruptedException e) {
```

```
        e.printStackTrace();
```

```
    }
```

```
    System.out.println("I have return");
```

```
    return number;
```

```
}
```

```
public void setNumber(int number) {
```

```
    this.number = number;
```

```
    cdLatch.countDown();
```

Генерация одного события

```
    System.out.println(cdLatch.getCount() + " left");
```

```
}
```

```
}
```

Поток, который вызовет этот метод, будет заблокирован, пока другие потоки не сгенерируют 5 событий

Вывод на экран того, сколько еще событий должно произойти

## Поток, который будет захватывать описанный выше класс

```
package com.gmail.tsa;

public class SingleThread implements Runnable {

    private SynchClass sClass;

    public SingleThread(SynchClass sClass) {
        super();
        this.sClass = sClass;
    }

    @Override
    public void run() {
        sClass.getNumber();
    }
}
```

Получение ссылки на синхронизируемый объект в конструкторе класса

Вызов метода синхронизированного объекта

Алгоритм работы данного проекта таков: поток попытается вызвать метод **getNumber()**, но будет заблокирован до наступления 5 событий, и будет ожидать их выполнения. Другой поток 5 раз вызовет метод **setNumber()**, тем самым генерируя по одному событию за вызов (событие генерируется вызовом метода **countDown()**). Как только будет сгенерировано 5 событий, ожидающий поток активизируется и выполниться до конца.

# Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        SynchronClass sClass = new SynchronClass();
```

```
        SingleThread sTh = new SingleThread(sClass);
```

```
        Thread th = new Thread(sTh);  
        th.start();
```

```
        for (int i = 0; i < 5; i++) {  
            sClass.setNumber(12);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }
```

```
    }  
    I am waiting  
    4 left  
    3 left  
    2 left  
    1 left  
    0 left  
    I have return
```

Создание объекта синхронизированного класса

Создание потока для захвата объекта

В главном потоке вызываем метод генерирующий событие 5 раз. Для разблокирования ожидающего потока.

Вывод на консоль. Как видите, поток действительно ожидает выполнения 5 событий, после чего выполняет свою работу.

## CyclicBarrier

CyclicBarrier — синхронизаторы, использующиеся при условии, что потоки должны остановить свое выполнение до тех пор, пока N потоков также не подойдут к указанной точке выполнения. Т.е. это - своеобразный барьер, в который «упрутся» потоки, пока их не наберется достаточное их количество.

Метод	Название
<code>int await()</code>	Блокирует поток, пока остальные потоки не достигнут барьера.
<code>int await(long time, TimeUnit unit)</code>	Блокирует поток, пока остальные потоки не достигнут барьера.

Конструкторы:

- `CyclicBarrier(int threadNumber)` — блокирует потоки по достижению барьера `threadNumber` потоков;
- `CyclicBarrier(int threadNumber, Runnable object)` — блокирует потоки по достижению барьера `threadNumber` потоков и потом запускает на выполнение `object`;



# Пример использования CyclicBarrier

```
package com.gmail.tsa;
```

```
import java.util.concurrent.BrokenBarrierException;
```

```
import java.util.concurrent.CyclicBarrier;
```

```
public class SinchClass {
```

```
    private CyclicBarrier cBar = new CyclicBarrier(5);
```

Создание на 5 потоков

```
    private int number;
```

```
    public SinchClass(int number) {
```

```
        super();
```

```
        this.number = number;
```

```
    }
```

Каждый поток, вызвавший этот метод, снимает одну блокировку, вызывая await()

```
    public void setNumber(int number) {
```

```
        try {
```

```
            System.out.println(Thread.currentThread().getName() + " set - wait");
```

```
            cBar.await();
```

```
            Thread.sleep(1000);
```

```
        } catch (InterruptedException | BrokenBarrierException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
        System.out.println(Thread.currentThread().getName() + " set - go");
```

```
        this.number = number;
```

```
    }
```

```
    public int getNumber() {
```

```
        try {
```

```
            System.out.println(Thread.currentThread().getName() + " get - wait");
```

```
            cBar.await();
```

```
            Thread.sleep(1000);
```

```
        } catch (BrokenBarrierException | InterruptedException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
        System.out.println(Thread.currentThread().getName() + " get - go");
```

```
        return number;
```

```
    }
```

```
}
```

## Поток для захвата объекта

```
package com.gmail.tsa;

public class SingleThread implements Runnable {
    private SinchClass sClass;

    public SingleThread(SinchClass sClass) {
        super();
        this.sClass = sClass;
    }

    @Override
    public void run() {
        sClass.setNumber(10);
    }
}
```

Т.е. поток попросту попытается вызвать один из синхронизируемых методов этого объекта.

## Главный класс проекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        SinchClass sClass = new SinchClass(5);
```

```
        for (int i = 0; i < 4; i++) {  
            Thread thread = new Thread(new SingleThread(sClass));  
            thread.start();  
        }
```

```
        System.out.println(sClass.getNumber());
```

```
    }
```

```
}
```

4 Потока заблокируется на объекте, так как для преодоления барьера нужно 5.



Главный поток будет 5 потоком на барьере. Он и разблокирует все остальные потоки

## Exchanger

Exchanger — применяется для упрощения обмена данными между двумя потоками. Принцип работы таков: он ожидает, когда потоки, между которыми нужно совершить обмен данными, выполнят метод `exchange()`. Как только этот метод будет вызван, будет произведен обмен данными.

Метод	Описание
<code>V exchange (V date)</code>	Производит обмен данных, сохраненных в буфере <code>date</code> . Поток блокируется до той поры, пока второй метод не вызовет свою версию метода.
<code>V exchange(V date, long time, TimeUnit unit)</code>	Производит обмен данными в буфере <code>date</code> . Поток блокируется или до вызова аналогичного метода другим потоком, или на <code>time</code> — единиц времени, выраженных с помощью <code>unit</code>

Внимание! Класс `Exchanger` является обобщенным. Как следствие, при его создании требуется явно указывать тип данных, хранящихся в буфере.

# Пример использования Exchanger

```
package com.gmail.tsa;
```

```
import java.util.concurrent.Exchanger;
```

```
public class SingleThreadOne implements Runnable {
```

```
    private Exchanger<String> exChanger;
```

```
    private String text;  
    private long time;
```

```
    public SingleThreadOne(Exchanger<String> exChanger, String text, long time) {  
        super();  
        this.exChanger = exChanger;  
        this.text = text;  
        this.time = time;  
    }
```

```
@Override
```

```
    public void run() {  
        System.out.println(Thread.currentThread().getName() + " - My text - " + this.text);
```

```
        try {
```

```
            Thread.sleep(time);
```

```
            this.text = exChanger.exchange(text);
```

```
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }
```

```
        System.out.println(Thread.currentThread().getName() + " - New text - " + this.text);
```

```
    }  
}
```

И хотя каждый поток будет спать разное время, метод обмена синхронизированный. Поэтому он заблокирует первый поток до, тех пор, пока этот метод не вызовет и второй.

Exchanger будет использоваться для синхронизированного обмена данными типа String

Вызов синхронизированного метода для обмена данными

## Главный класс проекта

```
package com.gmail.tsa;  
  
import java.util.concurrent.Exchanger;  
  
public class Main {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub
```

Exchanger для обмена

```
        Exchanger<String> exChanger = new Exchanger<>();
```

```
        SingleThreadOne stOne = new SingleThreadOne(exChanger, "Hello", 1000);  
        SingleThreadOne stTwo = new SingleThreadOne(exChanger, "World", 3000);
```

```
        Thread thOne = new Thread(stOne);  
        Thread thTwo = new Thread(stTwo);  
  
        thOne.start();  
        thTwo.start();
```

Потоки, которые будут обмениваться данными

Создание и запуск потоков

```
Thread-1 - My text - World  
Thread-0 - My text - Hello  
Thread-1 - New text - Hello  
Thread-0 - New text - World
```

Вывод на консоль: как вы видите - два потока произвели обмен данными.

## Итоги урока

Синхронизаторы — объекты, предназначенные для управления потоками в зависимости от внешних условий.

Одним из низкоуровневых средств синхронизации является интерфейс Lock с объектами условий, которые он генерирует.

Однако существуют и более высокоуровневые средства синхронизации:

- Semaphore
- CountdownLatch
- CyclicBarrier
- Exchanger

И хотя действие этих средств синхронизации легко реализовать с помощью низкоуровневых средств синхронизации, они значительно упрощают разработку многопоточного ПО.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 1008 — 1018
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала — Java . Том 1. Основы. Девятое издание. стр. 774-803



# Java OOP

## (java.lang.Object)

**Object** — суперкласс для всех классов. Поэтому методы этого класса доступны во всех его подклассах.

Благодаря этому выстраивается вертикальная иерархическая структура наследования, что дает возможность написания стандартных методов, которые присущи всем объектам в Java.

## Методы класса java.lang.Object

Метод	Описание
Object clone()	Создает новый объект, копирующий исходный
boolean equals(Object объект)	Проверяет равенство объектов
void finalize()	Вызывается перед удалением объекта
final Class <?> getClass()	Получает объект класса Class, который описывает вызывающий объект
int hashCode()	Возвращает хеш-код
final void notify()	Прерывает выполнение потока, ожидающего вызывающего объекта
final void notifyall()	Прерывает выполнение всех потоков, ожидающего вызывающего объекта
String toString()	Возвращает строку описывающую объект
final void wait()	Ожидает завершения выполнения другого потока
final void wait(long ms)	Ожидает завершения выполнения другого потока в ms
final void wait(long ms, int ns)	Ожидает завершения выполнения другого потока в ms и ns

## Метод toString()

Возвращает описание объекта в виде строки. Вызывается для символьных потоков (например, при попытке вывести в консоль объект). Можно переопределить.

```
package com.gmail.tsa;
```

Пример переопределения метода toString

```
public class Rectangle {
```

```
    double length;  
    double width;
```

```
    Rectangle(double length, double width){  
        this.length=length;  
        this.width=width;  
    }
```

Метод переопределен и теперь выводит информацию в нужном виде

```
@Override
```



```
    public String toString(){  
        return (" Rectangle "+length+" , "+width="  
            "+width);  
    }
```

```
}
```

## Метод **finalize()**

Метод `Object.finalize()` вызывается перед уничтожением объекта в памяти сборщиком мусора.

```
package com.gmail.tsa;

public class Rectangle {

    double length;
    double width;

    Rectangle(double length, double width){
        this.length=length;
        this.width=width;
    }
    @Override
    public String toString(){
        return (" Rectangle "+length+" , "+width);
    }
    public void finalize(){
        System.out.println("Rectangle destroyed!!");
    }
}
```



Этот метод выполнится перед удалением объекта

# Пример использования метода finalize()

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Rectangle a=new Rectangle(4,5);
```

```
        System.out.println(a);
```

← Использование метода toString()

```
        int n = 10;
```

```
        for( ;(n-- > 0);){
```

```
            new Rectangle(3,4);
```

← Создается 10 объектов без ссылки

```
        }
```

```
        System.gc();
```

← Принудительный вызов сборщика мусора

```
    }
```

```
}
```

## Object.hashCode() – хэш код объекта.

Хеширование — преобразование по детерминированному алгоритму входного массива данных произвольной длины в выходную битовую строку фиксированной длины.

Такие преобразования также называются хеш-функциями или функциями свертки, а их результаты называют хешем, хеш-кодом или сверткой сообщения.

По умолчанию для вычисления хеш-кода также используется и адрес объекта в памяти. Поэтому по умолчанию хеш-код одинаков только в случае, если два объекта по сути являются двумя ссылками на одни и те же данные. При переопределении для одинаковых объектов значение hashCode() тоже должно быть одинаковым!

## Пример использования стандартного метода hashCode()

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Rectangle a=new Rectangle(4,5);
```

Вывод на экран hash кода объекта

```
        System.out.println(a.hashCode());
```

```
    }
```

```
}
```

! Этот метод также можно переопределить под свои нужды. Обычно это применяется для структур данных, которые для хранения данных используют хеш-таблицы.

## Использование метода `equals`

Метод `equals` используется для сравнения объектов. Однако по умолчанию сравниваются ссылки на объект, а не содержимое объектов.

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Rectangle a=new Rectangle(4,5);  
        Rectangle b=new Rectangle(4,5);
```

Создаются два объекта с одинаковыми свойствами

```
        System.out.println(a.equals(b));
```



Метод вернет **false**, так как сравниваются ссылки, а они разные

**!** Для сравнения внутреннего содержимого нужно переопределить метод `equals`.



## Правила, которые должны выполняться при переопределении метода equals

### **public boolean equals(Object obj)**

- для любого ненулевого значения  $x$ ,  $x.equals(x)$  должен возвращать true.
- для любых ненулевых значений  $x$  и  $y$ ,  $x.equals(y)$  должен возвращать true тогда и только тогда, когда  $y.equals(x)$  возвращает true.
- для любых ненулевых значений  $x$ ,  $y$ ,  $z$ , если  $x.equals(y)$  возвращает true и  $y.equals(z)$  возвращает true, то  $x.equals(z)$  должна вернуть true.
- для любых ненулевых значений  $x$  и  $y$ , множественные вызовы  $x.equals(y)$  последовательно возвращают true или false если информация, которая используется при их сравнении не меняется.
- Для любых ненулевых  $x$ ,  $x.equals(null)$  должен вернуть false.

## Пример переопределения метода equals(Object a)

```
package com.gmail.tsa;

public class Rectangle {

    double length;
    double width;

    Rectangle(double length, double width){
        this.length=length;
        this.width=width;
    }

    @Override
    public String toString(){
        return (" Rectangle "+length+ " , "+width+ );
    }

    public void finalize(){
        System.out.println("Rectangle destroyed!!");
    }

    @Override
    public boolean equals(Object a){
        if(this==a) return true;
        else if(a==null) return false;
        else if(this.getClass()!=a.getClass()) return false;
        Rectangle b=(Rectangle)a;
        if((b.length==this.length) && (b.width==this.width)) return true;
        else return false;
    }
}
```

Переопределенный метод equals() для сравнения по содержимому

## Пример использования переопределенного метода equals

```
package com.gmail.tsa;  
  
public class Main {  
    public static void main(String[] args) {  
        Rectangle a=new Rectangle(4,5);  
        Rectangle b=new Rectangle(4,5);  
        System.out.println(a.equals(b));  
    }  
}
```

Теперь **equals** вернет **true**, так как  
объекты сравниваются по свойствам.

## О взаимосвязи hashCode() и equals()

Имеется связь между hash-кодом и сравнением. Если два объекта одинаковы (с пользовательской, логической точки зрения), то их hash-коды также должны быть одинаковы. Следовательно, если переопределяется метод сравнения, то также должен переопределяться метод для получения hash-кода.

! Переопределили equals() - переопределите и hashCode()

Это правило связано с тем, что разные классы-контейнеры используют разные методы для определения эквивалентности объекта. Т.е., например, списки используют equals и наборы hashCode. Отсюда следует требование, что эти методы должны быть взаимно однозначными.

# Пример переопределения hashCode() и equals()

```
package com.gmail.tsa;
```

```
public class Rectangle {
```

```
    double length;  
    double width;
```

```
    Rectangle(double length, double width) {  
        this.length = length;  
        this.width = width;  
    }
```

```
    @Override  
    public String toString() {  
        return (" Rectangle " + "length= " + length + " , " + "width= " + width);  
    }
```

```
    public void finalize() {  
        System.out.println("Rectangle destroyed!!");  
    }
```

```
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        long temp;  
        temp = Double.doubleToLongBits(length);  
        result = prime * result + (int) (temp ^ (temp >>> 32));  
        temp = Double.doubleToLongBits(width);  
        result = prime * result + (int) (temp ^ (temp >>> 32));  
        return result;  
    }
```

Теперь и хеш-код вычисляется на основании свойств класса

```
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        Rectangle other = (Rectangle) obj;  
        if (Double.doubleToLongBits(length) != Double.doubleToLongBits(other.length))  
            return false;  
        if (Double.doubleToLongBits(width) != Double.doubleToLongBits(other.width))  
            return false;  
        return true;  
    }  
}
```

Сравнение по свойству класса  
(длина и ширина)

## Реализация интерфейса Cloneable и создание копии объекта

Для создания копии объекта (а не еще одной ссылки на объект) нужно реализовать интерфейс супер класса Object — **Cloneable**. Для этого необходимо переопределить метод

- **protected Object clone()**

Правила, по которым работает метод clone()

- `x.clone() != x` -> true (т.к. разные объекты)
- `x.clone().getClass() == x.getClass()` -> true (одинаковый класс)
- `x.clone().equals(x)` -> true (реже false)

**Внимание!** При клонировании ссылочные объекты внутри класса нужно клонировать отдельно. В противном случае они будут ссылаться на одни и те же данные. Т.е. будет реализовано так называемое поверхностное клонирование.

## Пример класса реализующего интерфейс Cloneable

```
package com.gmail.tsa;

public class Rectangle implements Cloneable{
    double length;
    double width;
    Rectangle(double length,double width){
        this.length=length;
        this.width=width;
    }
    @Override
    public String toString(){
        return (" Rectangle "+length+ " , "+width+ );
    }
    public void finalize(){
        System.out.println("Rectange destroyed!!");
    }
    @Override
    public boolean equals(Object a){
        if(this==a) return true;
        else if(a==null) return false;
        else if(this.getClass()!=a.getClass()) return false;
        Rectangle b=(Rectangle)a;
        if((b.length==this.length) && (b.width==this.width)) return true;
        else return false;
    }
    @Override
    public Rectangle clone(){
        try{
            return (Rectangle) super.clone();
        }
        catch(CloneNotSupportedException e){
            return null;
        }
    }
}
```

Использование метода суперкласса Object

Переопределение метода clone()

Исключение, которое может вызвать метод

## Пример создания копии объекта

```
package com.gmail.tsa;
```

```
public class Main {
```

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

```
        Rectangle a=new Rectangle(4,5);
```

```
        Rectangle b=new Rectangle(4,5);
```

```
        Rectangle c=a.clone();
```

```
        a.length=3;
```

```
        System.out.println(a);  
        System.out.println(c);
```

```
    }
```

```
}
```

Копирование объекта a в объект c

Изменение свойства объекта a

Вывод на экран свойств объектов

Как можно будет видеть, свойства объектов стали одинаковыми, т. е. Вы получили клон объекта.



## Получение объекта Class с помощью метода getClass()

Для любого объекта в java в памяти создается переменная класса Class, в которой инкапсулируется описание класса (члены класса, реализуемые интерфейсы и т. д.). Получить ссылку на такую переменную можно вызвав для объекта метод getClass().

Также можно получить ссылку на класс используя имя класса и его свойство class. Например, Integer.class.

Так как для одного класса используемого, в вашем проекте генерируется только одна переменная класса Class, то их можно сравнивать с помощью оператора ==, что дает возможность однозначного определения, к какому классу принадлежит объект.

## Пример использования метода getClass()

```
package com.gmail.tsa;
```

```
import java.util.Scanner;
```

```
public class Main {
```

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

```
        Scanner sc = new Scanner(System.in);
```

```
        System.out.println(check(sc, Scanner.class));
```

```
        System.out.println(check(sc, int.class));
```

```
    }
```

```
    public static boolean check(Object obj, Class<?> classOne) {
```

```
        Class<?> classTwo = obj.getClass();
```

```
        if (classOne == classTwo) {  
            return true;
```

```
        }  
        return false;
```

```
    }
```

```
}
```

Результат true, так как sc действительно принадлежит классу Scanner



Результат false, так как sc не принадлежит классу Scanner



Получение переменной типа Class для объекта



Если класс объекта, который был первым параметром, совпадает с классом второго, то вернем true, а в противном случае false



# Сериализация

**Сериализация**-это процесс сохранения состояния объекта в последовательность байт.

**Десериализация**-это процесс восстановления объекта из этой последовательности.

Для указания возможности выполнения сериализации класс должен реализовать интерфейс **Serializable**.

Он не содержит никаких методов, а просто указывает на то, что класс и его подклассы могут быть сериализованны.

Переменные объявленные как `transient` (вы явно указываете что эти свойства не нужно сериализовать) и статические переменные не сохраняются средствами сериализации. Обычно это используют для того, чтобы не передавать конфиденциальные данные, например пароли. Их сначала шифруют, а потом уже подвергают процедуре сериализации вручную.

## Если стандартные средства сериализации недостаточны

Если по каким-то причинам стандартного метода сериализации недостаточно, то тогда нужно реализовывать интерфейс **Externalizable**

Интерфейс **Externalizable** определяет два метода, которые нужно переопределить:

- `void readExternal(ObjectInput входной поток) throws IOException, ClassNotFoundException;`
- `void writeExternal(ObjectOutput выходной поток) throws IOException;`

# Средства для сериализации и десериализации объектов

Для выполнения сериализации используются интерфейсы и реализующие их классы, связанные с входными и выходными потоками объектов.

Интерфейсы :

- **ObjectOutput** — сериализация
- **ObjectInput** - десериализация

Классы:

- **ObjectOutputStream** - сериализация
- **ObjectInputStream** - десериализация

## Интерфейс OutputStream

`void writeObject(Object Объект)` — метод записи объекта в поток

Метод	Описание
<code>void close()</code>	Закрывает вызывающий поток.
<code>void flush()</code>	Очищает все буферы.
<code>void write (byte буфер[])</code>	Записывает массив байт в вызывающий поток.
<code>void write (byte буфер[],int смещение, int кол-во байт)</code>	Записывает кол-во байт из массива в буфер, начиная с буфер[смещение].
<code>void write(int b)</code>	Записывает одиночный байт в вызывающий поток.
<code>void writeObject(Object Объект)</code>	Записывает объект в вызывающий поток.

## Класс ObjectOutputStream реализует интерфейс ObjectOutputStream

Конструктор:

ObjectOutputStream(OutputStream выходной поток)

Метод	Описание
void close()	Закрывает вызывающий поток.
void flush()	Очищает все буферы.
void write (byte буфер[])	Записывает массив байтов вызывающий поток.
void write (byte буфер[],int смещение, int кол-во байт)	Записывает кол-во байт из массива буфер начиная со буфер[смещение].
void write(int b)	Записывает одиночный байт в вызывающий поток.
final void writeObject(Object Объект)	Записывает объект в вызывающий поток.
void writeDouble(double d)	Записывает double переменную.
void writeInt(int i)	Записывает int переменную.
void writeString(String s)	Записывает строку.
void writeBoolean(Boolean b)	Записывает boolean переменную.

## Интерфейс ObjectInput

Object readObject() — метод для считывания объекта

Метод	Описание
int available()	Возвращает кол-во доступных байтов.
void close()	Закрывает поток.
int read()	Возвращает целочисленное представление доступного байта. При окончании возвращает -1.
int read(byte буфер[])	Пытается прочитать до размера буфера байт в буфер, и возвращает кол-во прочитанных. При окончании вернет -1.
int read (byte буфер[],int смещение, int кол-во байт)	Пытается прочитать до размера буфера байт в буфер начиная со смещения, и возвращает кол-во прочитанных. При окончании вернет -1.
Object readObject()	Читает объект.
Long skip(long numBytes)	Пропускает указанное кол-во байт, вернет кол-во действительно пропущенных.



## Класс `ObjectInputStream` реализует интерфейс `ObjectInput`

Конструктор:

`ObjectInputStream(InputStream входной поток)`

Метод	Описание
<code>int available()</code>	Возвращает кол-во доступных байтов
<code>void close()</code>	Закрывает поток
<code>int read()</code>	Возвращает целочисленной представление доступного байта. При окончании возвращает -1.
<code>int read(byte буфер[])</code>	Пытается прочитав до размера буфера байт в буфер, и возвращает кол-во прочитанных. При окончании вернет -1.
<code>int read (byte буфер[],int смещение, int кол-во байт)</code>	Пытается прочитав до размера буфера байт в буфер начиная со смещения, и возвращает кол-во прочитанных. При окончании вернет -1.
<code>final Object readObject()</code>	Читает объект.
<code>long skip(long numBytes)</code>	Пропускает указанное кол-во байт, вернет кол-во действительно пропущенных.
<code>double readDouble()</code>	Читает и возвращает double.

# Пример использования сериализации.

Часть I

Создадим класс, который описывает группу людей, и сохраним его на диск с последующим чтением.

```
package com.gmail.tsa;  
  
import java.io.Serializable;
```

Если класс имеет классы свойства, то они тоже должны поддерживать сериализацию

```
public class Human implements Serializable{
```

```
    private static final long serialVersionUID = 1L;
```

```
    String name;
```

```
    int age;
```

```
    char sex;
```

```
    Human(String name, int age, char sex){
```

```
        this.name=name;
```

```
        this.age=age;
```

```
        this.sex=sex;
```

```
    }
```

```
    @Override
```

```
    public String toString(){
```

```
        return (" Name - "+name+" Age - "+ age+ " Sex - "+  
                String.valueOf(sex));
```

```
    }
```

```
}
```

Для сериализуемого объекта вычисляется его контрольная сумма. Однако при наличии такого статического члена она не вычисляется, а берется заданная.

Переопределение метода toString()

## Описание класса — группы людей

Часть II

```
package com.gmail.tsa;
```

```
import java.io.*;
```

Класс поддерживает сериализацию

```
public class Group implements Serializable{  
    private Human [] group;  
    private static final long serialVersionUID = 1L;  
    Group(){  
        group=new Human[0];  
    }
```

Добавление человека в группу

```
    public void addHuman(String name, int age, char sex){  
        Human[] c= new Human[group.length+1];  
        System.arraycopy(group, 0,c , 0, group.length);  
        c[c.length-1]=new Human(name,age,sex);  
        group=c;  
    }
```

```
    public void printgroup(){  
        for(Human k:group){  
            System.out.println(k);  
        }  
    }
```

Вывод информации о всей группе

```
}
```

# Пример сериализации и десериализации объектов

```
package com.gmail.tsa;
import java.io.*;
public class Main {
    public static void main(String[] args) {
```

```
        Group PN121=new Group();
        PN121.addHuman("Alexander", 21, 'М');
        PN121.addHuman("Alexey", 18, 'М');
        PN121.addHuman("Katia", 18, 'Ж');
```

Создание и наполнение 1-го объекта

```
        System.out.println();
        System.out.println("Вывод данных исходного объекта");
        System.out.println();
        PN121.printgroup();
```

Создание объектного потока для записи

```
        try(ObjectOutputStream OOS=new ObjectOutputStream(new FileOutputStream("fil"))){
```

```
            OOS.writeObject(PN121);
```

Запись объекта в файл

```
        }
        catch(IOException e){
            System.out.println("ERROR save group !!!");
        }
```

Создание объектного потока - чтение

```
        Group KT321=null;
```

```
        try (ObjectInputStream OIS=new ObjectInputStream(new FileInputStream("fil"))){
```

```
            KT321=(Group)OIS.readObject();
```

Считывание объекта

```
        }
        catch(IOException | ClassNotFoundException e){
            System.out.println("ERROR load group !!!");
        }
```

```
        System.out.println();
        System.out.println("Вывод данных считанного объекта");
        System.out.println();
        KT321.printgroup();
```

Убедимся, что объект верно считан

```
    }
}
```

## Итоги урока

Класс `Object` неявно является суперклассом для всех классов в `java`. Благодаря этому факту методы, которые в нем определены, становятся доступными для любого пользовательского класса.

Для переопределения чаще всего используются такие методы:

- `ToString()`;
- `Equals()`;
- `HashCode()`;

Также существует возможность сериализации и диссериализации объектов с использованием классов `ObjectInputStream` и `ObjectOutputStream`.

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 233 — 235
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала Java . Том 1. Основы. Девятое издание. стр. 218-232

## Домашнее задание

- 1) Используя стандартные методы сериализации создайте мини-базу данных для работы с группами студентов (возможность записи и чтения базы из файла по запросу пользователя).
- 2) Создайте класс-контейнер типа стек (класс в который можно добавлять и удалять объекты других классов, только в вершину стека), в который можно сохранять объекты произвольного типа. Должен быть метод добавления элемента в стек, получение с удалением элемента из стека, и просто получение элемента из вершины из стека. Должна быть реализована работа с «черным списком» классов (смотри ниже). Если объект который добавляется в стек принадлежит классу из «черного списка», то добавление такого объекта запрещено.
- 3) Для описанного выше стека создайте класс «Черный список», в котором будут описаны классы объектов которые нельзя добавлять в стек. Должна быть возможность добавления классов в черный список, проверка объекта на то, что класс, к которому он принадлежит, принадлежит или не принадлежит к классам в черном списке.

# Маршалинг и демаршалинг объектов в XML и JSON

## Дополнительный материал к лекции «Класс `java.lang.Object`»

Маршалинг (от англ. *marshal* — упорядочивать) в информатике - процесс преобразования информации (данных, двоичного представления объекта), хранящейся в оперативной памяти, в формат, пригодный для хранения или передачи. Обычно применяется тогда, когда информацию (данные, объекты) необходимо передавать между различными частями одной программы или от одной программы к другой.

Противоположный процесс называется демаршалингом.

**Составил: Цымбалюк А.Н.**



## Описание формата хранения данных XML

**XML**-eXtensible Markup Language — расширяемый язык разметки). Рекомендован Консорциумом Всемирной паутины (W3C). Спецификация XML описывает XML-документы и частично описывает поведение XML-процессоров (программ, читающих XML-документы и обеспечивающих доступ к их содержимому). XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчеркиванием нацеленности на использование в Интернете.

**XML** — это описанная в текстовом формате иерархическая структура, предназначенная для хранения любых данных. Визуально структура может быть представлена как дерево элементов. Элементы XML описываются тегами.

Формально для java это означает, что вы можете взять объект пользовательского класса, и на его основе сгенерировать запись в текстовом файле (т. е. Обычный текст) - маршалинг и наоборот, можно получив набор строк, получить объект - демаршалинг.

# Структура XML документа - Пролог

В общем случае XML документ состоит из **пролога** и **корневого элемента**

## Пролог:

- **Объявление XML** - Объявление XML указывает версию языка, на которой написан документ. В первой (1.0) версии языка использование объявления не было обязательным, в последующих версиях оно обязательно. Таким образом, версия языка определяется из объявления, и если объявление отсутствует, то принимается версия 1.0. Объявление может также содержать информацию о кодировке документа и «оставаться ли документу со своим собственным DTD, или с подключённым».
  - Пример - `<?xml version="1.1" encoding="UTF-8" ?>`
  - Пример - `<?xml version="1.0" encoding="UTF-8" standalone="yes"?>` - тут указано что определение документа будет подключено извне.
- **Объявление типа документа** - Для объявления типа документа существует специальная инструкция `!DOCTYPE`. Она позволяет задать при помощи языка DTD, какие в документ входят элементы, каковы их атрибуты, какие сущности могут использоваться.
- **Инструкция обработки** - Инструкции обработки, позволяют размещать в документе инструкции для приложений.
- **Комментарий** - Комментарии не относятся к символьным данным документа. Комментарий начинается последовательностью `<!--` и заканчивается последовательностью `-->`, внутри не может встречаться комбинация символов `--`. Символ `&` не используется внутри комментария в качестве разметки.

# Структура XML документа — Корневой элемент

## Корневой элемент

- Элемент является понятием логической структуры документа. Каждый документ содержит один или несколько элементов. Границы элементов представлены начальным и конечным тегами. Имя элемента в начальном и конечном тегах элемента должно совпадать. Элемент может быть также представлен тегом пустого, то есть не включающего в себя другие элементы и символьные данные, элемента.
- **Тег** - Теги могут быть трех видов: открывающий (<name>), закрывающий (</name>) и пустой (name/)
- **Элемент** - Элемент начинается с открывающего тега и заканчивается соответствующим закрывающим (или состоит только из пустого тега). Он также может содержать прочие элементы, которые будут называться потомками.
- **Атрибут** - Атрибут представляет собой пару «имя/значение», которая располагается в открывающем или закрывающем теге.

# Пример XML документа

```
<?xml version="1.0"?>
```

← Пролог

**Element** — описывает сущность (по сути объект)

```
<catalog>
```

```
<book>
```

```
<author>Gambardella, Matthew</author>
```

```
<title>XML Developer's Guide</title>
```

```
<genre>Computer</genre>
```

```
<price>44.95</price>
```

```
<publish_date>2000-10-01</publish_date>
```

```
</book>
```

**Text** — значение некоторой сущности (значение свойства объекта)

**Attribute** - атрибут (дополнительные данные о свойстве)

**catalog** — это корневой элемент, потому что в этом элементе содержатся все остальные элементы

```
<book>
```

```
<author>Ralls, Kim</author>
```

```
<title>Midnight Rain</title>
```

```
<genre>Fantasy</genre>
```

```
<price currency="USD">5.95</price>
```

```
<publish_date>2000-12-16</publish_date>
```

```
</book>
```

```
</catalog>
```

← Элемент book — это книга со свойствами

В данном документе описывается каталог книг, каждым элементом каталога является книга, книга же в свою очередь состоит из элементов author, title, genre, price, publish\_date

## Спецификации XML в Java

**JAXP** (Java Architecture for XML Processing — архитектура Java для обработки XML) — это низкоуровневая спецификация (JSR 206), которая дает возможность очень гибко обрабатывать XML, а также позволяет использовать SAX, DOM или XSLT. Этот API также применяется JAXB и StAX.

Спецификация **JAXB** обеспечивает набор API-интерфейсов и аннотаций для представления XML-документов как артефактов Java, что позволяет работать с соответствующими объектами Java. JAXB (JSR 222) обрабатывает демаршалинг документов XML в объекты и наоборот.

**StAX** (Streaming API for XML — потоковый API для XML) версии 1.0 (JSR 173) представляет собой API для чтения и записи XML-документов. Основным направлением его деятельности является использование преимуществ древоподобных API (анализаторов DOM) и API, основанных на событиях (анализаторов SAX). Первые позволяют получить произвольный, неограниченный доступ к документу, в то время как вторые занимают меньше памяти и предъявляют меньше требований к процессору.

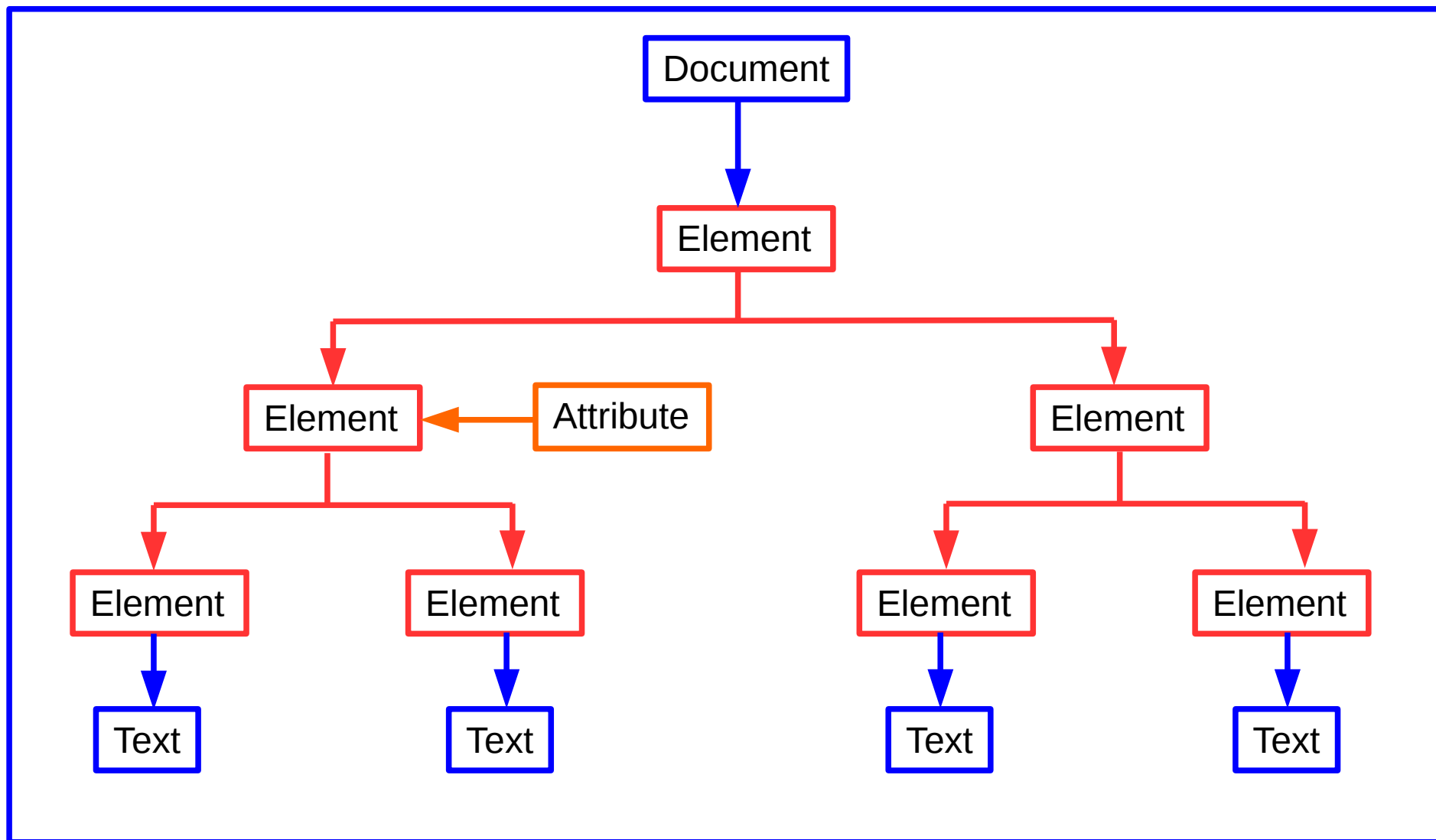
## DOM — модель документа и анализаторы DOM в java

DOM (от англ. Document Object Model — «объектная модель документа») — это не зависящий от платформы и языка программный интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML-, XHTML- и XML-документов, а также изменять содержимое, структуру и оформление таких документов. Модель DOM не накладывает ограничений на структуру документа. Любой документ известной структуры с помощью DOM может быть представлен в виде дерева узлов, каждый узел которого представляет собой элемент, атрибут, текстовый, графический или любой другой объект. Узлы связаны между собой отношениями «родительский-дочерний».

Организация «World Wide Web Consortium» (W3C) занимается стандартизацией DOM-модели. В 2016 году последняя версия стандарта была DOM-3. Ссылка на эту спецификацию - <https://www.w3.org/TR/DOM-Level-3-Core/>

DOM представляет собой дерево в виде специальных объектов Node. Каждый Node соответствует своему XML-тегу. Каждый Node содержит полную информацию о том, что это за тег, какие он имеет атрибуты, какие дочерние узлы содержит внутри себя и так далее. На самой вершине этой иерархии находится Document.

## Схематическое изображение DOM - дерева



Document – контейнер для всех узлов в дереве. Он также называется корнем документа.

## org.w3c.dom.Node – родительский интерфейс для всех узлов в DOM дереве.

Interface	nodeName	nodeValue	attributes
Attr	same as Attr.name	same as Attr.value	null
CDATASection	"#cdata-section"	same as CharacterData.data, the content of the CDATA Section	null
Comment	"#comment"	same as CharacterData.data, the content of the comment	null
Document	"#document"	null	null
DocumentFragment	"#document-fragment"	null	null
DocumentType	same as DocumentType.name	null	null
Element	same as Element.tagName	null	NamedNodeMap
Entity	entity name	null	null
EntityReference	name of entity referenced	null	null
Notation	notation name	null	null
ProcessingInstruction	same as ProcessingInstruction.target	same as ProcessingInstruction.data	null
Text	"#text"	same as CharacterData.data, the content of the text node	null

Атрибуты nodeName, nodeValue и attributes включены в качестве механизма для получения информации на узле без приведения до конкретного полученного интерфейса. В тех случаях, когда нет никакого очевидного отображения этих атрибутов для конкретного NODETYPE (например, nodeValue для элемента или атрибуты для комментария), это возвращает null.



## Получение данных из XML документа разбором DOM дерева

Порядок разбора XML документа:

- 1) Создать объект класса — **DocumentBuilderFactory**
- 2) На его основе нужно создать парсер для документа - **DocumentBuilder**
- 3) Создать документ из строки или потока ввода — **Document**
- 4) Получить корневой элемент класса — **Element**
- 5) Выполнить разбор древовидной структуры дерева.

Постановка задачи: Описать классы Book, Catalog и, используя стандартные средства Java, выполнить демаршалинг из xml файла. Файл взят из примера XML файла, описанного выше.

```
package com.gmail.tsa;

import java.util.Date;

public class Book {
    public enum Currency {
        UAH, USD, EUR
    };
    private String author;
    private String title;
    private String genre;
    private double price;
    private Date publishDate;
    private Currency cur = Currency.UAH;

    public Book(String author, String title, String genre, double price, Date publishDate) {
        super();
        this.author = author;
        this.title = title;
        this.genre = genre;
        this.price = price;
        this.publishDate = publishDate;
    }
    public Book() {
        super();
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public String getTitle() {
        return title;
    }
}
```

```
public void setTitle(String title) {
    this.title = title;
}
public String getGenre() {
    return genre;
}
public void setGenre(String genre) {
    this.genre = genre;
}
public double getPrice() {
    return price;
}
public void setPrice(double price) {
    this.price = price;
}
public Date getPublishDate() {
    return publishDate;
}
public void setPublishDate(Date publishDate) {
    this.publishDate = publishDate;
}
public Currency getCur() {
    return cur;
}
public void setCur(String cur) {
    this.cur = Currency.valueOf(cur);
}
@Override
public String toString() {
    return "Book [author=" + author + ", title=" + title + ", genre=" + genre + ", price=" +
        price + " " + cur.toString() + ", publishDate=" + publishDate + "]";
}
}
```

## Класс Catalog

```
package com.gmail.tsa;

import java.util.ArrayList;

public class Catalog {
    private ArrayList<Book> listBook = new ArrayList<>();

    public void addBook(Book book) {
        if (book == null) {
            throw new IllegalArgumentException("Null");
        }
        listBook.add(book);
    }

    public ArrayList<Book> getListBook() {
        return new ArrayList<Book>(listBook);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Book book : listBook) {
            sb.append(book).append(System.lineSeparator());
        }
        return sb.toString();
    }
}
```

Т.е. это класс-контейнер для объектов типа Book — для него и будет организовываться процесс демаршалинга.

```
package com.gmail.tsa;
```

# Класс для демаршалинга из XML файла

Часть 1

```
import java.io.File;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

```
public class CatalogXMLWorker {
    public static Catalog loadCatalogFromXMLFile(File file) {
        Catalog catalog = new Catalog();
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

Создание DocumentBuilderFactory

```
    try {
```

```
        DocumentBuilder builder = factory.newDocumentBuilder();
```

Создание documentBuilder на основе объекта DocumentBuilderFactory

```
        Document document = builder.parse(file);
```

```
        Element root = document.getDocumentElement();
```

Получение корневого элемента

```
        NodeList books = root.getChildNodes();
```

Получение списка дочерних узлов

```
        for (int i = 0; i < books.getLength(); i++) {
```

```
            Node node = books.item(i);
```

Получение i — го узла из списка

```
            if (node.getNodeType() == Node.ELEMENT_NODE) {
```

```
                Element element = (Element) node;
                Book book = getBookFromNode(element);
```

```
                if (book != null) {
                    catalog.addBook(book);
                }
            }
        }
```

Проверка: если текущий узел не текст, то преобразуем его в Element и вызываем метод для его разбора

```
    } catch (Exception e) {
        return null;
    }
```

```
    return catalog;
}
```

```
private static Book getBookFromNode(Element bookElement) {
```

```
    if (!bookElement.getTagName().equals("book")) {  
        return null;  
    }
```

Проверяем что элементу отвечает тег book

```
    String author = bookElement.getElementsByTagName("author").item(0).getTextContent();  
    String title = bookElement.getElementsByTagName("title").item(0).getTextContent();  
    String genre = bookElement.getElementsByTagName("genre").item(0).getTextContent();  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
    String dateText = bookElement.getElementsByTagName("publish_date").item(0).getTextContent();  
    Date publishDate = new Date();  
    try {  
        publishDate = sdf.parse(dateText);  
    } catch (ParseException e) {  
        System.out.println("Error load publish_date");  
    }
```

Получаем дочерние узлы по именам их тегов и получаем данные хранящиеся в них.

```
    Node priceNode = bookElement.getElementsByTagName("price").item(0);
```

```
    double price = Double.valueOf(priceNode.getTextContent());
```

```
    Book book = new Book(author, title, genre, price, publishDate);
```

```
    if (priceNode.hasAttributes()) {  
        Node currencyNode = priceNode.getAttributes().item(0);  
        String curr = currencyNode.getNodeValue();  
        book.setCur(curr);  
    }  
    return book;
```

Получаем узел price. Так как у него могут быть атрибуты, исследуем его подробнее

```
}
```

```
}
```

## Главный класс проекта

```
package com.gmail.tsa;

import java.io.File;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Catalog catalog = CatalogXMLWorker.loadCatalogFromXMLFile(new
        File("catalog.xml"));
        System.out.println(catalog);
    }

}
```

```
Book [author=Gambardella, Matthew, title=XML Developer's Guide, genre=Computer, price=44.95 UAH,
publishDate=Sun Oct 01 00:00:00 EEST 2000]
Book [author=Ralls, Kim, title=Midnight Rain, genre=Fantasy, price=5.95 USD, publishDate=Sat Dec 16
00:00:00 EET 2000]
```



Вывод на консоль. Как вы видите, демаршалинг из XML файла прошел успешно.

## Некоторые наиболее часто употребляемые методы при демаршалинге XML

```
NodeList books = root.getChildNodes();
```

Вернет список дочерних узлов

```
node.getNodeType() == Node.ELEMENT_NODE
```

Определить тип узла

```
bookElement.getElementsByTagName("author")
```

Получить список узлов у тега с заданным именем

```
item(0)
```

Получить узел из списка узлов по индексу

```
getTextContent();
```

Получить текстовое содержимое элемента

```
hasAttributes()
```

Узнать, есть ли атрибут у узла

```
priceNode.getAttributes().item(0);
```

Получение атрибута в виде элемента по индексу



## Запись данных в XML документ с использованием DOM дерева

Порядок записи XML документа:

- 1) Создать объект класса — **DocumentBuilderFactory**
- 2) На его основе нужно создать парсер для документа - **DocumentBuilder**
- 3) Создать документ — **Document**
- 4) Создать корневой элемент класса — **Element**
- 5) Создать древовидную структуру документа.
- 6) Создать объект класса — **TransformerFactory**
- 7) На его основе создать преобразователь документа в текст — **Transformer**
- 8) Установить необходимые параметры преобразователю
- 9) Создать набор данных на основе узла документа - **DOMSource**
- 10) Создать поток для преобразователя - **StreamResult**
- 11) Сохранить документ

Постановка задачи: Описать классы Book, Catalog и, используя стандартные средства Java, выполнить маршалинг в xml файл.

# Метод создание Element на основе объекта

```
private static Element elementFromBook(Book book, Document document) {
```

```
    Element bookElement = document.createElement("book");
```

Создаем элемент  
ответственный за книгу

```
    Element author = document.createElement("author");  
    author.setTextContent(book.getAuthor());
```

```
    Element title = document.createElement("title");  
    title.setTextContent(book.getTitle());
```

Пример создания элемента  
и установки его свойств

```
    Element genre = document.createElement("genre");  
    genre.setTextContent(book.getGenre());
```

```
    Element price = document.createElement("price");  
    price.setTextContent("" + book.getPrice());
```

```
    if (!book.getCur().name().equals("UAH")) {  
        price.setAttribute("currency", book.getCur().name());  
    }
```

При необходимости  
добавляем атрибут

```
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
    Element publishDate = document.createElement("publish_date");  
    publishDate.setTextContent(sdf.format(book.getPublishDate()));
```

```
    bookElement.appendChild(author);  
    bookElement.appendChild(title);  
    bookElement.appendChild(genre);  
    bookElement.appendChild(price);  
    bookElement.appendChild(publishDate);
```

Добавление созданных подэлементов как  
дочерних к элементу bookElement

```
    return bookElement;
```

```
}
```

# Метод маршалинга catalog в XML файл

```
public static void saveToXML(Catalog catalog, String fileName) {
```

```
    DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();  
    try {
```

```
        DocumentBuilder docBuilder = documentBuilderFactory.newDocumentBuilder();  
        Document document = docBuilder.newDocument();
```

```
        Element root = document.createElement("catalog");  
        document.appendChild(root);  
        for (Book book : catalog.getListBook()) {  
            Element bookEl = elementFromBook(book, document);  
            root.appendChild(bookEl);  
        }
```

Создание DocumentBuilder и Document на его основе

```
        TransformerFactory traF = TransformerFactory.newInstance();  
        Transformer transformer = traF.newTransformer();
```

Создание преобразователя

```
        DOMSource source = new DOMSource(document);
```

Создаем источник данных из документа

```
        StreamResult stRes = new StreamResult(fileName);
```

Создаем поток для сохранения в файл

```
        transformer.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "yes");  
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");  
        transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
```

```
        transformer.transform(source, stRes);
```

Запись

```
    } catch (ParserConfigurationException | TransformerException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

Установка параметров записи

## Подробнее о маршалинге в XML (подготовка документа)

Т.е. сначала нужно создать документ с древовидной структурой

```
DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
try {
    DocumentBuilder docBuilder = documentBuilderFactory.newDocumentBuilder();
    Document document = docBuilder.newDocument();
```

Создать на его основе корневой элемент

```
Element root = document.createElement("catalog");
```

Важно! Прикрепить корневой элемент к документу

```
document.appendChild(root);
```

Создать нужное количество подэлементов и установить их свойства

```
Element title = document.createElement("title");
title.setTextContent(book.getTitle());
```

Прикрепляем созданные элементы к родительским, создавая древовидную структуру

```
bookElement.appendChild(title);
```

## Подробнее о маршалинге в XML (сохранение документа)

### Создание преобразователя

```
TransformerFactory traF = TransformerFactory.newInstance();  
Transformer transformer = traF.newTransformer();
```

### Создание источника данных из документа

```
DOMSource source = new DOMSource(document);
```

### Создание потока для сохранения

```
StreamResult stRes = new StreamResult(fileName);
```

### Установка требуемых свойств преобразователя

```
transformer.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "yes");  
transformer.setOutputProperty(OutputKeys.INDENT, "yes");  
transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
```

### Преобразование и сохранение

```
transformer.transform(source, stRes);
```

## Java Architecture for XML Binding

Java Architecture for XML Binding (JAXB) позволяет Java разработчикам ставить в соответствие Java классы и XML представления. JAXB предоставляет две основные возможности: маршализрование Java объектов в XML и наоборот, то есть демаршализация из XML обратно в Java объект. Другими словами, JAXB позволяет хранить и извлекать данные в памяти в любом XML-формате, без необходимости выполнения определенного набора процедур загрузки и сохранения XML.

Т.о. есть возможность каждому объекту пользовательского класса поставить в соответствие запись в XML документе, и наоборот, можно создать объект пользовательского класса на основе такой записи.

Это дает возможность использовать механизм аннотирования для построения отображения класса Java в запись в XML документе.

## Связывание

JAXB API, определенный в пакете `javax.xml.bind`, предоставляет набор интерфейсов и классов для создания XML-документов и генерации классов Java. Другими словами, он связывает две модели. Фреймворк среды выполнения JAXB реализует операции маршалинга и демаршалинга.

## Структура JAXB

Пакет	Описание
<code>javax.xml.bind</code>	Фреймворк связывания среды выполнения, имеющий возможность выполнять операции маршалинга, демаршалинга и проверки
<code>javax.xml.bind.annotation</code>	Аннотации для настройки преобразований между программой Java и XML-данными
<code>javax.xml.bind.annotation.adapters</code>	Классы-адаптеры JAXB
<code>javax.xml.bind.attachment</code>	Позволяет выполнять маршалинг для оптимизации хранения двоичных данных и демаршалинг корня документа, содержащего форматы двоичных данных
<code>javax.xml.bind.helpers</code>	Содержит частичные стандартные реализации некоторых интерфейсов <code>javax.xml.binding</code>
<code>javax.xml.bind.util</code>	Предоставляет полезные вспомогательные классы

В центре JAXB API находится класс [`javax.xml.bind.JAXBContext`](#). Этот абстрактный класс управляет связыванием между XML-документами и объектами Java, поскольку он предоставляет:

- `Unmarshaller`, который преобразует XML-документ в граф объектов и, возможно, проверяет XML;
- `Marshaller`, который принимает граф объектов и трансформирует его в документ XML;



## Список аннотаций в JAXB

Аннотация	Тип
@XmlRootElement	Представляет аннотацию, необходимую любому классу для связывания в качестве корневого элемента XML
@XmlElement	Преобразует нестатический атрибут или геттер без модификатора transient в XML-элемент
@XmlAttribute	Преобразует атрибут или геттер к XML-атрибуту примитивного типа
@XmlElements	Действует как контейнер для нескольких аннотаций @XmlElement
@XmlEnum	Преобразует перечисление к представлению XML
@XmlEnumValue	Идентифицирует константу-перечисление
@XmlAccessorType	Управляет необходимостью преобразования атрибутов или геттеров (FIELD, NONE, PROPERTY, PUBLIC_MEMBER)
@XmlID	Идентифицирует ключевое поле элемента XML (или типа String), которое может быть использовано для ссылки на элемент с использованием аннотации @XmlIDREF (концепции XML Schema ID и IDREF)
@XmlIDREF	Преобразует свойство в схеме в XML IDREF
@XMLList	Преобразует свойство в список
@XmlMimeType	Идентифицирует текстовое представление типа MIME
@XmlNs	Идентифицирует пространство имен XML
@XmlSchema	Преобразует имя пакета в пространство имен XML
@XmlTransient	Информирует JXB, что не нужно связывать атрибут (аналогичен ключевому слову transient в Java или аннотации @Transient в JPA)
@XmlType	Аннотирует класс как комплексный тип в схеме XML
@XMLValue	Позволяет преобразовать класс в простое содержимое или тип схемы

Т.е. вам нужно проаннотировать требуемые свойства класса в соответствии с типом



```
package com.gmail.tsa;

import java.text.SimpleDateFormat;
import java.util.Date;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

## Класс Train

```
@XmlRootElement(name = "train")
```

Аннотация для указания того, что это - объект

```
public class Train {
    private String from;
    private String to;
    private Date departureTime;

    public Train(String from, String to, Date departureTime) {
        super();
        this.from = from;
        this.to = to;
        this.departureTime = departureTime;
    }

    public Train() {
        super();
    }
}
```

```
@XmlElement
public String getFrom() {
    return from;
}
```

Аннотация на геттер свойства для создания подэлемента

```
public void setFrom(String from) {
    this.from = from;
}
```

```
@XmlElement
public String getTo() {
    return to;
}

public void setTo(String to) {
    this.to = to;
}
```

```
@XmlElement
@XmlJavaTypeAdapter(TrainsDateFormatter.class)
public Date getDepartureTime() {
    return departureTime;
}
```

Обратите внимание на это поле. Оно снабжено адаптером

```
public void setDepartureTime(Date departureTime) {
    this.departureTime = departureTime;
}
```

```
@Override
public String toString() {
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy - hh:mm");
    return from + " - " + to + " - " + sdf.format(departureTime);
}
```

```
}
```

## Класс адаптер для даты

```
package com.gmail.tsa;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class TrainsDateFormatter extends XmlAdapter<String, Date> {
    private SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy - hh:mm");

    @Override
    public Date unmarshal(String v) throws Exception {
        Date date = new Date();
        try {
            date = sdf.parse(v);
        } catch (ParseException e) {
            System.out.println(e);
        }
        return date;
    }

    @Override
    public String marshal(Date v) throws Exception {
        return sdf.format(v);
    }
}
```

Объект из строки

Строка из объекта

Если нет встроенного преобразования объекта в строку, стоит использовать класс-адаптер. В нем необходимо переопределить, как генерировать строку на основе объекта и как генерировать объект на основе строки.

```
package com.gmail.tsa;
```

## Класс расписание поездов

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

```
import javax.xml.bind.annotation.XmlElement;
```

```
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement
```

Аннотация для указания того, что это-объект

```
public class Trains {
```

```
    @XmlElement(name = "train")
```

```
    private ArrayList<Train> listTrain = new ArrayList<>();
```

```
    public Trains() {
```

```
        super();
```

```
    }
```

```
    public void addTrain(Train train) {
```

```
        listTrain.add(train);
```

```
    }
```

```
    public ArrayList<Train> getListTrains() {
```

```
        return new ArrayList<>(listTrain);
```

```
    }
```

```
@Override
```

```
    public String toString() {
```

```
        StringBuilder sb = new StringBuilder();
```

```
        listTrain.stream()
```

```
            .forEachOrdered(n -> sb.append(n).append(System.lineSeparator()));
```

```
        return sb.toString();
```

```
    }
```

```
}
```

Аннотация на свойство для создания подэлемента

# Класс для маршалинга и демаршалинга списка поездов

```
package com.gmail.tsa;
```

```
import java.io.File;
```

```
import javax.xml.bind.JAXBContext;
```

```
import javax.xml.bind.JAXBException;
```

```
import javax.xml.bind.Marshaller;
```

```
import javax.xml.bind.Unmarshaller;
```

```
public class JAXBWorker {
```

```
    public static void saveToXMLFile(Trains trains, File file) {  
        try {  
            JAXBContext jaxbC = JAXBContext.newInstance(Trains.class);  
            Marshaller marSh = jaxbC.createMarshaller();  
            marSh.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);  
            marSh.marshal(trains, file);  
        } catch (JAXBException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }
```

Маршалинг списка поездов

```
    public static Trains loadFromXMLFile(File file) {  
        JAXBContext jaxbC = null;  
        try {  
            jaxbC = JAXBContext.newInstance(Trains.class);  
            Unmarshaller unmarshaller = jaxbC.createUnmarshaller();  
            Trains trains = (Trains) unmarshaller.unmarshal(file);  
            return trains;  
        } catch (JAXBException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        return null;  
    }
```

Демаршалинг списка поездов

```
}
```

## Маршалинг подробнее

```
public static void saveToXMLFile(Trains trains, File file) {  
    try {
```

```
        JAXBContext jaxbC = JAXBContext.newInstance(Trains.class);
```

Создание JAXBContext с указанием класса

```
        Marshaller marSh = jaxbC.createMarshaller();
```

Создание объекта Marshaller

```
        marSh.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

Указание формата маршалинга

```
        marSh.marshal(trains, file);
```

Маршалинг

```
    } catch (JAXBException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }
```

```
}
```

## Демаршалинг подробнее

```
public static Trains loadFromXMLFile(File file) {
```

```
    try {
```

```
        JAXBContext jaxbC = JAXBContext.newInstance(Trains.class);
```

Создание JAXBContext с указанием класса

```
        Unmarshaller unmarshaller = jaxbC.createUnmarshaller();
```

Создание объекта Unmarshaller

```
        Trains trains = (Trains) unmarshaller.unmarshal(file);
```

Демаршалинг

```
        return trains;
```

```
    } catch (JAXBException e) {
```

```
        // TODO Auto-generated catch block
```

```
        e.printStackTrace();
```

```
    }
```

```
    return null;
```

```
}
```

## Главный класс проекта

```
package com.gmail.tsa;

import java.io.File;
import java.util.Date;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Trains trains = new Trains();
        trains.addTrain(new Train("London", "Paris", new Date(1234555556789L)));
        trains.addTrain(new Train("London", "Monako", new Date(1235555556789L)));
        trains.addTrain(new Train("London", "Lviv", new Date(1234655556789L)));
        trains.addTrain(new Train("London", "Kiev", new Date(1234755556789L)));

        JAXBWorker.saveToFile(trains, new File("trains.xml"));

        Trains trainsTwo = JAXBWorker.loadFromFile(new File("trains.xml"));

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

Маршалинг в файл

Демаршалинг из файла

## Результат маршалинга

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<trains>
  <train>
    <departureTime>13/02/2009 - 10:05</departureTime>
    <from>London</from>
    <to>Paris</to>
  </train>
  <train>
    <departureTime>25/02/2009 - 11:52</departureTime>
    <from>London</from>
    <to>Monako</to>
  </train>
  <train>
    <departureTime>15/02/2009 - 01:52</departureTime>
    <from>London</from>
    <to>Lviv</to>
  </train>
  <train>
    <departureTime>16/02/2009 - 05:39</departureTime>
    <from>London</from>
    <to>Kiev</to>
  </train>
</trains>
```

Как можно видеть, маршалинг списка поездов прошел успешно



## Основные сведения о JSON

Нотация объектов JavaScript (**JSON**) представляет собой легкий формат обмена данными, то есть менее подробный и более удобочитаемый, чем XML. Он часто используется для сериализации и передачи структурированных данных через сетевое соединение между сервером и веб-приложением.

В качестве альтернативы XML JSON непосредственно используется в коде JavaScript на веб-страницах. Это основная причина прибегать к JSON, а не к другому представлению данных.

JSON-объекты можно сериализовать в JSON, который в конечном итоге будет иметь менее сложную структуру, чем XML. Заключив значение переменной в фигурные скобки, вы указываете, что ее значение является объектом. Внутри объекта мы можем объявить любое количество свойств, используя пары "имя": "значение", разделенные запятыми. Чтобы получить доступ к информации, хранящейся в JSON, можно просто сослаться на объект и имя свойства.

**Документы JSON.** JSON является текстовым, независимым от языка форматом, который использует набор соглашений для представления простых структур данных. Структуры данных в JSON просты для понимания и очень похожи на структуры данных в Java. JSON может представлять четыре примитивных типа (число, строка, двоичное значение и null) и два структурированных (объекты и массивы).

## Соглашение о представлении данных в JSON

Терминология	Описание
Число	Число в JSON очень похоже на таковое в Java, за исключением того, что в JSON не могут применяться восьмеричные и шестнадцатеричные значения
Строка	Строка — это последовательность из нуля или более символов Unicode, ограниченная двойными кавычками.
Значение	Значение в JSON может быть представлено в одном из следующих форматов: строка в двойных кавычках, число, двоичное значение, объект или массив
Массив	Массив — это упорядоченный набор значений. Квадратные скобки ([,]) обозначают начало и конец массива. Значения массива разделены запятыми (,) и могут быть объектами
Объект	JSON и Java имеют одинаковое определение объекта. В JSON объект — это неупорядоченный набор пар «имя/значение». Фигурные скобки ({,}) обозначают начало и конец объекта. Пары «имя/значение» в JSON разделяются запятыми (,) и представляют атрибуты POJO в Java

Двоеточие в JSON используется в качестве разделителя имен, а запятая — в качестве разделителя значений. Корректные данные JSON представляют собой сериализованный объект или массив структур данных, которые могут быть вложенными. Например, объект JSON может содержать массив JSON.

## Пример JSON - документа

```
{  
  "name": "Alexander",  
  "lastName": "Is",  
  "age": 35,  
  "groupName": "PN121"  
}
```

Т.е. значение свойств объекта хранится в виде «имя свойства» : «значение»

Имя свойства

Значение свойства

"name": "Alexander"

# GSON — библиотека от Google, значительно упрощающая работу с JSON

Для относительно простой работы с форматом JSON можно использовать библиотеку от Google GSON. Загрузить можно отсюда <https://github.com/google/gson>

Основным классом библиотеки является класс **Gson**. Для того, чтобы создать экземпляр класса, нужно воспользоваться одним из двух способов:

```
Gson gson = new Gson();
```

```
Gson gson = new GsonBuilder().create();
```

Первый способ создаст объект класса с настройками по умолчанию, а второй способ позволит применить некоторые настройки.

Основные методы, которые используются для сериализации и десериализации java-объектов, называются toJson и fromJson.

Ниже будет описан класс для сериализации и десериализации в формат JSON, и пример использования этой библиотеки.

## Класс Студент

```
package com.gmail.tsa;

public class Student {
    private String name;
    private String lastName;
    private int age;
    private String groupName;

    public Student(String name, String lastName, int age, String groupName) {
        super();
        this.name = name;
        this.lastName = lastName;
        this.age = age;
        this.groupName = groupName;
    }

    public Student() {
        super();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getGroupName() {
        return groupName;
    }

    public void setGroupName(String groupName) {
        this.groupName = groupName;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", lastName=" + lastName + ", age=" + age + ", groupName=" + groupName + "];"
    }
}
```

## Класс для сериализации и десериализации в JSON формат

```
package com.gmail.tsa;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonIOException;
import com.google.gson.JsonSyntaxException;
```

```
public class JSONWorker {
```

```
    public static void saveToJSONFile(Student student, File file) {
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        String gsonSt = gson.toJson(student);
        try (PrintWriter pw = new PrintWriter(file)) {
            pw.println(gsonSt);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
```

Сериализация

```
    public static Student loadFromJSON(File file) {
        Gson gson = new Gson();
        Student student = null;
        try {
            student = gson.fromJson(new FileReader(file), Student.class);
        } catch (JsonSyntaxException | JsonIOException | FileNotFoundException e) {
            e.printStackTrace();
        }
        return student;
    }
```

Десериализация

## Сериализация в JSON

```
public static void saveToJSONFile(Student student, File file) {
```

```
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
```

```
    String gsonSt = gson.toJson(student);
```

```
    try (PrintWriter pw = new PrintWriter(file)) {  
        pw.println(gsonSt);  
    } catch (IOException e) {  
        System.out.println(e);  
    }
```

Запись строки в файл

```
}
```

Создание объекта класса Gson. Обратите внимание, объект создается с настройкой параметров `.setPrettyPrinting()` для более красивого вывода данных

Далее получаем строку, в которую и будут записаны данные объекта в формате JSON

Т.е. для сериализации объекта достаточно просто подставить его в метод `toJson()` и явной необходимости указывать имена полей для сохранения с помощью аннотаций нет.

## Десериализация из JSON

```
public static Student loadFromJSON(File file) {
```

```
Gson gson = new Gson();
```

Создание объекта класса JSON

```
Student student = null;
```

```
try {
```

Десериализация

Откуда

Какой класс объекта

```
student = gson.fromJson(new FileReader(file), Student.class);
```

```
} catch (JsonSyntaxException | JsonIOException |  
FileNotFoundException e) {  
    e.printStackTrace();  
}
```

```
return student;
```

```
}
```

Т.е. десериализация средствами JSON также не очень сложна. Нужно указать строку или источник строк для десериализации и класс объекта, создаваемого из файла.



## Главный класс проекта

```
package com.gmail.tsa;  
  
import java.io.File;  
  
public class Main {  
    public static void main(String[] args) {
```

Сериализация



```
Student student = new Student("Alexander", "Ts", 35, "PN121");  
JSONWorker.saveToJSONFile(student, new File("a.gson"));
```

```
Student studentOne = JSONWorker.loadFromJSON(new File("a.gson"));  
System.out.println(studentOne);
```



Десериализация

```
}  
  
}
```

# Работа с массивами в GSON

## Массивы

Работа с массивами не отличается от работы с обычными классами

```
package com.gmail.tsa;

import java.util.Arrays;
import com.google.gson.Gson;

public class Main {

    public static void main(String[] args) {

        int[] array = { 3, 5, 7 };

        Gson gson = new Gson();

        String result = gson.toJson(array);

        System.out.println(result);

        int[] arrayOne = gson.fromJson(result, int[].class);

        System.out.println(Arrays.toString(arrayOne));

    }

}
```

Запись массива в формат JSON

Считывание

## Работа с Map в GSON

В общем случае сериализация Map происходит без проблем. Однако десериализация нуждается в дополнительном уточнении класса ключа и значения.

Для этого нужно указать переменную типа Type, созданную подобным образом:

Класс ключа



Класс ключа



```
Type type = new TypeToken<Map<Integer, String>>() {}.getType();
```

Потом нужно указать созданную переменную вторым параметром метода toJson()

## Пример сериализации и десериализации Map

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Type;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
import com.google.gson.Gson;
```

```
import com.google.gson.GsonBuilder;
```

```
import com.google.gson.reflect.TypeToken;
```

```
public class Main {
```

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

```
        Map<Integer, String> map = new HashMap<>();  
        map.put(1, "one");  
        map.put(2, "two");  
        map.put(3, "three");
```

Создание и заполнение Map

```
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
```

```
        String result = gson.toJson(map);
```

```
        System.out.println(result);
```

Указание типа ключа и значения

```
        Type type = new TypeToken<Map<Integer, String>>() {}.getType();
```

```
        Map<Integer, String> readMap = gson.fromJson(result, type);
```

```
        System.out.println(readMap);
```

Десериализация Map из строки. Type переменная это 2-й параметр этого метода

```
    }  
}
```

# Работа с Collection в GSON

Работа с Collection схожа на работу с Map. А именно, нужно указывать тип обобщенного класса Collection при десериализации.

```
package com.gmail.tsa;  
  
import java.util.ArrayList;  
  
import com.google.gson.Gson;  
import com.google.gson.reflect.TypeToken;  
  
public class Main {  
  
    public static void main(String[] args) {
```

```
        ArrayList<Integer> list = new ArrayList<>();  
        list.add(5);  
        list.add(7);  
        list.add(9);
```

Создание и заполнение List

```
        Gson gson = new Gson();  
        String result = gson.toJson(list);  
        System.out.println(result);
```

Сериализация

```
        ArrayList<Integer> listTwo = gson.fromJson(result, new  
        TypeToken<ArrayList<Integer>>() {}.getType());
```

```
        System.out.println(listTwo);
```

Десериализация. Второй параметр указывает на тип List

```
    }  
}
```

# Написание произвольного адаптера для сериализации и десериализации в GSON

Иногда нужно реализовать собственный механизм сериализации, десериализации в JSON. В таком случае пишется класс-адаптер. Для сериализации ему необходимо реализовать интерфейс JsonSerializer, а для десериализации - соответственно JsonDeserializer.

## Интерфейс - JsonSerializer

```
public JsonElement serialize(Custom src, Type type, JsonSerializerContext context);
```

## Интерфейс – JsonDeserializer

```
public Custom deserialize(JsonElement json, Type type, JsonDeserializationContext context)
```

Задача: Выполнить сериализацию и десериализацию объектов класса Train с использованием пользовательского класса - адаптера

## Класс Train

```
package com.gmail.tsa;

import java.util.Date;

public class Train {
    private String from;
    private String to;
    private Date departmentDate;

    public Train(String from, String to, Date departmentDate) {
        super();
        this.from = from;
        this.to = to;
        this.departmentDate = departmentDate;
    }
    public Train() {
        super();
    }
    public String getFrom() {
        return from;
    }
    public void setFrom(String from) {
        this.from = from;
    }
    public String getTo() {
        return to;
    }
    public void setTo(String to) {
        this.to = to;
    }
    public Date getDepartmentDate() {
        return departmentDate;
    }
    public void setDepartmentDate(Date departmentDate) {
        this.departmentDate = departmentDate;
    }
    @Override
    public String toString() {
        return "Train [from=" + from + ", to=" + to + ", departmentDate=" + departmentDate + "];"
    }
}
```

## Класс адаптер для сериализации и десериализации класса Train

```
package com.gmail.tsa;
```

```
import java.lang.reflect.Type;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import com.google.gson.JsonDeserializationContext;
import com.google.gson.JsonDeserializer;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParseException;
import com.google.gson.JsonSerializationContext;
import com.google.gson.JsonSerializer;
```

```
public class JSONTrainWorker implements JsonSerializer<Train>, JsonDeserializer<Train> {
    SimpleDateFormat sdf = new SimpleDateFormat("dd:MM:yyyy - hh:mm");
```

```
@Override
```

```
public Train deserialize(JsonElement arg0, Type arg1, JsonDeserializationContext arg2) throws
    JsonParseException {
    JsonObject jsonObject = arg0.getAsJsonObject();
    String from = jsonObject.get("from").getAsString();
    String to = jsonObject.get("to").getAsString();
    Date date = new Date();
    try {
        date = sdf.parse(jsonObject.get("department_time").getAsString());
    } catch (ParseException e) {
        e.printStackTrace();
    }
    return new Train(from, to, date);
}
```

Метод для десериализации

```
@Override
```

```
public JsonElement serialize(Train arg0, Type arg1, JsonSerializationContext arg2) {
    JsonObject jobject = new JsonObject();
    jobject.addProperty("from", arg0.getFrom());
    jobject.addProperty("to", arg0.getTo());
    jobject.addProperty("department_time", sdf.format(arg0.getDepartmentDate()));
    return jobject;
}
```

Метод для сериализации

```
}
```



## Главный класс проекта

```
package com.gmail.tsa;

import java.util.Date;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class Main {

    public static void main(String[] args) {
        Train train = new Train("London", "Paris", new Date());

        GsonBuilder gsBuilder = new GsonBuilder();

        gsBuilder.registerTypeAdapter(Train.class, new JSONTrainWorker());


        gsBuilder.setPrettyPrinting();

        Gson gson = gsBuilder.create();

        String result = gson.toJson(train);
        System.out.println(result);

        Train trainOne = gson.fromJson(result, Train.class);
        System.out.println(trainOne);
    }
}
```

Регистрация класса адаптера



Сериализация и десериализация средствами GSON



## Некоторые настройки GSON

Если нужно отключить сериализацию, десериализацию полей с заданными модификаторами

```
Gson gson = new GsonBuilder()  
    .excludeFieldsWithModifiers(Modifier.STATIC, Modifier.TRANSIENT, Modifier.VOLATILE)  
    .create();
```

Если нужно включить только поля, отмеченные аннотацией @Expose

```
GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();
```

Если нужно чтобы свойство класса сохранялось под именем отличным от его имени, то аннотируйте его @SerializedName("New\_name"), где вместо New\_name поставьте нужное вам название.

## Итоги урока

**Маршалинг** - процесс преобразования информации (данных, двоичного представления объекта), хранящейся в оперативной памяти, в формат, пригодный для хранения или передачи.

**Демаршалинг** — восстановление объекта из записанного ранее документа.

Наибольшее распространение получили два формата хранения документов **XML** - предоставляет более широкие возможности для управления содержимым, и **JSON** - который оличается простотой и легковесностью.

Для маршалинга, демаршалинга в XML документ можно использовать как прямую работу с DOM-деревом документа, так и более высокоуровневые интерфейсы работы посредством механизма аннотаций. Работа с DOM деревом дает возможность тонкой настройки документа и манипуляции с ним. Однако, если это не нужно, а нужна простота использования, то можно использовать интерфейсы стандарта **JAXB**.

**GSON** — внешняя библиотека от Google. Она позволяет работать исключительно легко с форматом JSON. Для этого просто нужно, чтобы ваш класс удовлетворял спецификации Java Bean. Стандартные средства сериализации и десериализации, при необходимости, можно снабдить дополнительной информацией для работы с массивами, Map, Collection. Также, реализуя встроенные интерфейсы этой библиотеки, легко реализовать собственные инструменты сериализации и десериализации.

## Дополнительная литература по теме данного урока.

- Энтони Гонсалвес. Изучаем Java EE 7. СПб.: Питер, 2014. стр. 467 — 497
- <https://google.github.io/gson/apidocs/com/google/gson/Gson.html>

# Java OOP

## Generics and Collection

Обобщение в Java (англ. generics) - это возможность обобщенного программирования, которая была добавлена в язык программирования Java в 2004 году как часть стандартной платформы J2SE 5.0.

Обобщение дает возможность создавать типы или методы таким образом, чтобы они могли оперировать различными типами данных, и при этом на этапе компиляции для типов обеспечивается соответствующий механизм безопасности.

Составил: Цымбалюк А.Н.

# Обобщенный класс

Обобщения — это параметризованные типы. Т.е. тип данных в этом случае сам является параметром. Классы, интерфейсы и методы, оперирующие параметризованными типами, называются обобщенными.

Для создания обобщенного класса используется конструкция вида:

```
class имя_класса <T> {...  
<T> - параметр типа
```

Параметр типа является своеобразной подстановкой для реального типа переменных, переданных классу, при работе программы. Если параметров типа несколько, то они перечисляются через запятую.

**!** Обобщения работают только с объектами

## Пример создания обобщенного класса

Параметр типа

```
package com.gmail.tsa;
```



```
public class Operation <T>{
```

```
    T obj;
```



Вместо параметра типа подставится реальный тип

```
    Operation(T a){  
        this.obj=a;  
    }
```

```
    void print(){  
        System.out.println(obj.toString());  
    }
```

```
}
```


## Пример использования обобщенного класса

```
package com.gmail.tsa;
```

```
public class Main {
```

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

Создание объекта обобщенного класса. Параметр типа - Integer




```
        Operation<Integer> a=new Operation<Integer> (100);
```

```
        a.print();
```

```
        Operation <String> b= new Operation<String>("HELLO");
```

```
        b.print();
```

Создание объекта обобщенного класса. Параметр типа - String



```
    }
```

```
}
```

Для указания значения типа используется запись <Параметр>



Фактически, после указания параметра, для текущего объекта стирается информация о типе, и остается только тот тип что был выбран в качестве параметра. Например для первого примера T заменяется на Integer.



## Ограниченные типы

Иногда возникает необходимость в ограничении типов (например, нужно использовать только числовые типы).

Для этого в описании типов можно использовать конструкции вида

<T extends суперкласс>

Тогда, вместо T, может быть поставлен только суперкласс, либо его наследники.

Вы также можете устанавливать ограничения по интерфейсу. В таком случае можно их объединять с помощью символа &.

<T extends суперкласс & интерфейс>

## Использование ограничения типов

```
package com.gmail.tsa;
```

```
public class Operation <T extends Number, V extends Number> {
```

```
    T obj;  
    V obj1;
```

```
    Operation(T a, V b){  
        this.obj=a;  
        this.obj1=b;  
    }
```

```
    void print(){  
        System.out.println(obj.toString());  
    }
```

```
    double sum(){  
        return obj.doubleValue()+obj1.doubleValue();  
    }
```

```
}
```

Ограничение только на числовые типы

У каждого подкласса Number есть метод doubleValue() и поэтому этот метод работает. А вот для Object он бы не сработал.

## Пример использования ограниченных обобщенных классов

```
package com.gmail.tsa;  
  
import java.math.BigInteger;  
  
public class Main {  
    public static void main(String[] args) {  
        Operation<Double, BigInteger> op = new Operation<>(34.5, new BigInteger("345"));  
        System.out.println(op.sum());  
    }  
}
```

Создание объекта обобщенного класса



Operation<Double, BigInteger> op = new Operation<>(34.5, new BigInteger("345"));

System.out.println(op.sum());

И хотя Double и BigInteger разные классы, они оба наследники Number, а следовательно метод их сложения выполнится корректно.

## Использование шаблонов аргументов

Можно использовать обобщенные аргументы для того или иного метода, что позволит не перегружать функции, а писать одну с обобщенными аргументами.

Для указания типа обобщенного аргумента достаточно указать в качестве аргумента

Конструкции типа

(Тип класса <?> имя переменной)

Тип класса - обобщенный класс

Если нужно ограничить тип, то указывается конструкция вида

(Тип\_класса <? extends супер\_класс>)

Однако, если нужно создать обобщенный метод в не обобщенном классе, то синтаксис объявления выглядит как:

<Обобщенные\_тип> тип\_возвращаемого\_значения Имя\_метода (параметры обобщенного типа)

# Пример создания обобщенного класса.

```
package com.gmail.tsa;
```

Описание обобщенного класса — есть обобщенные свойства E-типа

```
public class GroupAll<E> {  
    private E[] arr = (E[]) new Object[10];  
    private int i = 0;
```

Создание массива объектов E-типа. Внимание! Просто создать объект обобщенного типа нельзя. Поэтому создание возможно только через приведение Object.

```
    public void add(E st) {  
        for (int i = 0; i < arr.length; i++) {  
            if (arr[i] == null) {  
                arr[i] = st;  
                this.i++;  
                return;  
            }  
        }  
    }
```

Метод для работы со свойством обобщенного типа в обобщенном классе

```
    public int getI() {  
        return i;  
    }
```

Ограничение типов, с которыми может работать данный метод

```
    <T extends Number> double sum(T a, T b) {  
        return a.doubleValue() + b.doubleValue();  
    }
```

Метод работающий с аргументами обобщенного типа

```
    public boolean check(GroupAll<?> a) {  
        return (this.getI() == a.getI());  
    }
```

Метод, аргумент которого - объект обобщенного класса

```
}
```

## Пример использования обобщенного класса.

```
package com.gmail.tsa;
```

```
import java.math.BigInteger;
```

```
public class Main {
```

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

```
        GroupAll<BigInteger> gr = new GroupAll<>();  
        gr.add(new BigInteger("125"));  
        gr.add(new BigInteger("330"));
```

```
        System.out.println(gr.sum(100, new BigInteger("120")));
```


```
        GroupAll<String> gr2 = new GroupAll<>();  
        gr2.add("Hello");
```

```
        System.out.println(gr.check(gr2));
```


```
    }
```

```
}
```


Создание и использование объекта  
обобщенного класса



Использование метода с  
обобщенным аргументом



Использование метода работающего с  
объектами обобщенного класса



## Ограничение параметрических типов

У параметрических типов есть ряд ограничений:

- Нельзя создать новый объект обобщенного типа
- Статические члены класса не могут быть обобщенными
- Нельзя создать массив специфических для типа обобщений-ссылок
- Обобщенный класс не может быть унаследован от Throwable

## Коллекции

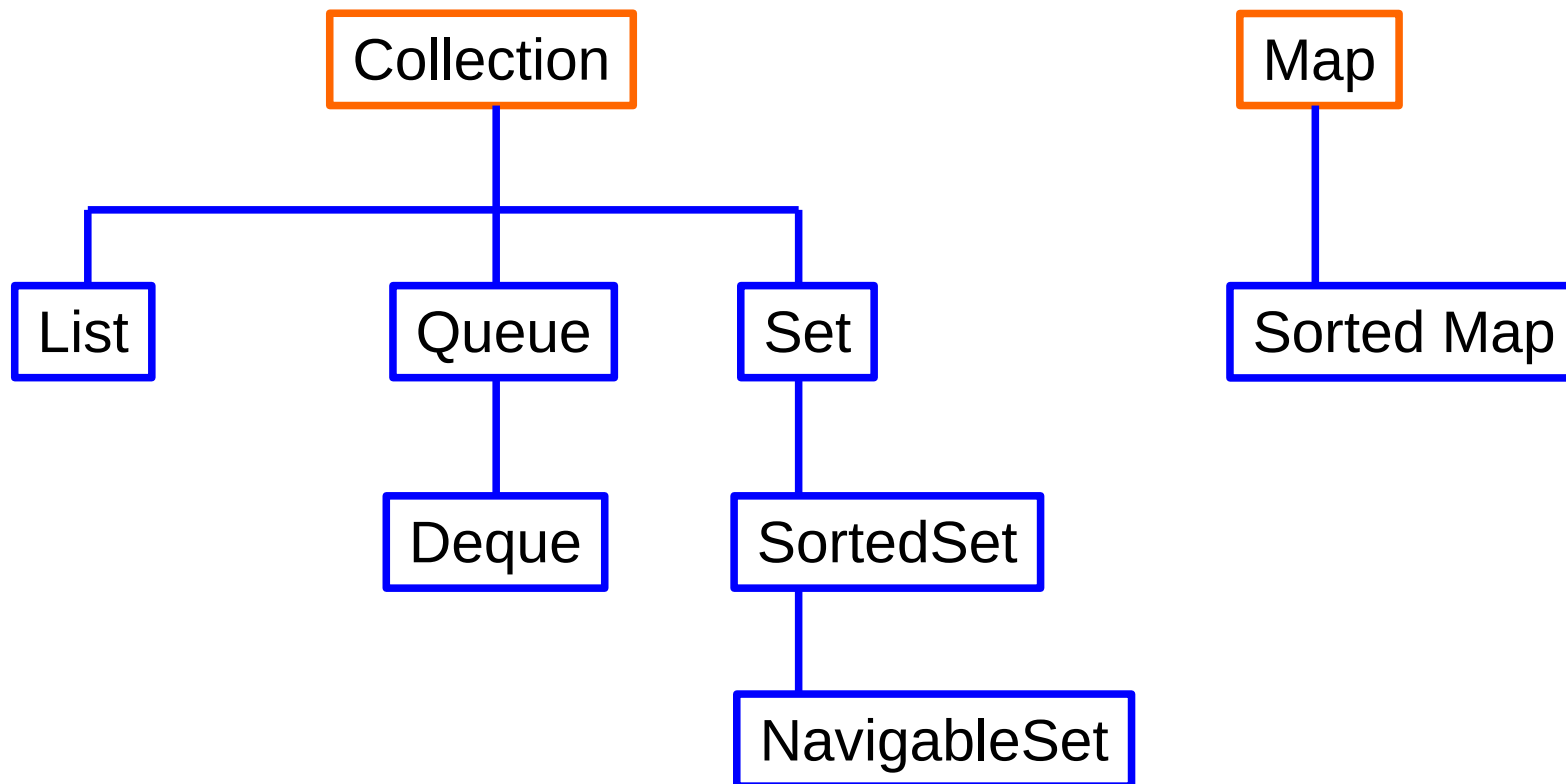
Коллекции или контейнеры — классы позволяющие хранить и производить операции над множеством объектов.

Каркас коллекций Collections Framework в Java стандартизирует способы управления группами объектов в прикладных программах. Коллекции не были частью исходной версии языка Java, но были внедрены в версии J2SE 1 .2.

Алгоритмы составляют другую важную часть каркаса коллекций. Алгоритмы оперируют коллекциями и определены в виде статических методов в классе Collections. Таким образом, они доступны всем коллекциям и не требуют реализации их собственной версии в каждом классе коллекции. Алгоритмы предоставляют стандартные средства для манипулирования коллекциями.



# Интерфейсы коллекций



Collection – корень иерархии коллекций.

Set – коллекция-множество, которая не может содержать одинаковые элементы.

List – список; может содержать одинаковые элементы; доступ по индексу.

Deque – двухсторонняя очередь; FIFO, LIFO.

Map – таблица типа «ключ – значение».

# Collection

Интерфейс Collection расширяет интерфейс Iterable т.е. все классы реализующие интерфейс Collection дают возможность перебора своих элементов.

Метод	Описание
<code>boolean add(E object)</code>	Добавляет объект к коллекции
<code>boolean addAll(Collection &lt;? extends E&gt; c</code>	Добавляет все элементы к коллекции
<code>void clear()</code>	Удаляет все элементы коллекции
<code>boolean contains( Object obj)</code>	Проверяет, является ли объект элементом коллекции
<code>Iterator &lt;E&gt; iterator()</code>	Возвращает итератор коллекции
<code>boolean equals(Collection &lt;?&gt; c)</code>	Проверяет эквивалентность коллекции и объекта
<code>int size()</code>	Возвращает размер коллекции
<code>Object[] toArray()</code>	Возвращает массив с элементами коллекции
<code>boolean removeAll(Collection &lt;?&gt; c)</code>	Удаляет все элементы «с» из коллекции
<code>int hashCode()</code>	Возвращает хеш код коллекции
<code>boolean retainAll(Collection &lt;?&gt; c)</code>	Удаляет все элементы кроме «с»
<code>boolean remove(Object obj)</code>	Удаляет элемент из коллекции

# List

Интерфейс, реализующий список - (в информатике, список англ. list) — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности — кортежа.

Метод	Описание
<code>void add(int index, E object)</code>	Вставляет объект на указанную позицию
<code>boolean addAll(int index, Collection &lt;? extends E&gt; c)</code>	Вставляет все элементы, начиная с указанной позиции
<code>E get(int index)</code>	Возвращает объект по позиции
<code>int indexOf(Object obj)</code>	Возвращает индекс 1-го объекта в списке
<code>int lastIndexOf(Object obj)</code>	Возвращает индекс последнего объекта в списке
<code>ListIterator&lt;E&gt; listiterator()</code>	Возвращает итератор для списка (указывает на начало списка)
<code>ListIterator&lt;E&gt; listiterator(int index)</code>	Возвращает итератор для списка (указывает на индекс)
<code>E remove(int index)</code>	Удаляет и возвращает элемент по индексу
<code>E set(int index, E obj)</code>	Присваивает объект элементу по индексу
<code>List&lt;E&gt; subList(int start,int end)</code>	Создает список из списка (по индексам)

## Класс ArrayList

Реализует интерфейс List. По сути, это - динамический массив с многими дополнительными методами.

### Конструктор

```
ArrayList()  
ArrayList(Collection <? extends E> c)  
ArrayList(int емкость)
```

Внимание! ArrayList не является списком в полном смысле этого слова. Он только реализует интерфейс List. На самом деле это - класс, внутри которого существует массив, дополненный методами для удобной работы с ним.

## Использование ArrayList

```
package com.gmail.tsa;
```

```
import java.util.*;
```

```
public class Main {
```

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

```
        ArrayList <Integer> a =new ArrayList<Integer>();
```

```
        a.add(34);  
        a.add(123);  
        a.add(45);
```

Добавление элементов

```
        System.out.println(Arrays.toString(a.toArray()));
```

```
        System.out.println(a.indexOf(123));
```

Поиск элемента в списке

```
        Iterator<Integer> itr=a.iterator();  
        System.out.println("Iterators ");
```

```
        for(;;itr.hasNext();){  
            System.out.println(itr.next());  
        }
```

Создание и использование итератора

```
    }
```

```
}
```

Создание ArrayList с  
содержим типом Integer

## Интерфейс Set и SortedSet, NavigableSet

Интерфейс Set определяет набор элементов не допускающих повторения, т.е. вставки элемента не произойдет, если такой уже имеется в наборе.

Интерфейс SortedSet расширяет интерфейс Set и объявляет набор отсортированный по возрастанию, не допускающих повторений элементов.

Интерфейс NavigableSet расширяет интерфейс SortedSet и объявляет набор отсортированный по возрастанию, не допускающих повторений элементов. В придачу реализовано извлечение элементов на основе ближайшего соответствия заданному значению, либо значениям.

## Методы интерфейса SortedSet

Методы	Описание
Comparator<? Super E> comparator()	Возвращает компаратор отсортированного набора
E first()	Возвращает первый элемент отсортированного набора
SortedSet<E> headSet(E end)	Возвращает объект интерфейса SortedSet, содержащий элементы из вызывающего набора, которые предшествуют end
E last()	Возвращает последний элемент отсортированного набора
SortedSet<E> subSet(E begin,E end)	Возвращает объект интерфейса SortedSet, содержащий элементы из вызывающего набора между begin и end
SortedSet<E> tailSet(E begin)	Возвращает объект интерфейса SortedSet, содержащий элементы из вызывающего набора, которые следуют за begin

## Методы интерфейса NavigableSet

Методы	Описание
E ceiling (E obj)	Ищет в наборе наименьший элемент «а», для которого «а»>=E
Iterator<E> descendingIterator()	Возвращает обратный итератор
NavigableSet<E> descendingSet()	Возвращает объект интерфейса равный обратной версии вызывающего набора
E floor(E obj)	Ищет в наборе наибольший элемент «а», для которого «а»<=E
NavigableSet<E> headSet( E top,boolean On)	Возвращает объект интерфейса, включающий все элементы вызывающего набора меньше top
E higher (E obj)	Ищет в наборе наименьший элемент «а», для которого «а»>E . Если объект найден, он возвращается, если нет, то null.
E lower(E obj)	Ищет в наборе наибольший элемент «а», для которого «а»<E . Если объект найден, он возвращается, если нет, то null.
E pollFirst()	Возвращает первый элемент из набора вместе с его удалением.
E pollLast()	Возвращает последний элемент из набора вместе с его удалением
NavigableSet<E> subset (E low, boolean Onlow, E top, boolean Ontop)	Возвращаются все элементы набора в диапазоне от low до top
NavigableSet<E> subset (E low, boolean Onlow)	Возвращает элементы больше low



## Класс HashSet

Класс HashSet реализует интерфейс Set. Он создает коллекцию объектов, использующих для хранения хеш-таблицу

### Конструктор

```
HashSet()  
HashSet(Collection <? extends E> c)  
HashSet(int емкость)  
HashSet(int емкость, float коэффициент заполнения)
```

Выгода от использования этого класса в том, что время выполнения методов add(), contains(), remove(), size() не зависит от размера коллекции.

## Класс LinkedHashSet

Класс LinkedHashSet расширяет класс HashSet. Он создает коллекцию объектов, использующих для хранения хеш-таблицу

### Конструктор

LinkedHashSet()

LinkedHashSet(Collection <? extends E> c)

LinkedHashSet(int емкость)

LinkedHashSet(int емкость, float коэффициент заполнения)

Отличие от HashSet в том, что элементы расположены в том же порядке, что и добавлялись.

## Класс TreeSet

Класс TreeSet расширяет класс AbstractSet и реализует интерфейс NavigableSet. Он создает коллекцию объектов, использующих для хранения хеш-таблицу и сортирует их.

### Конструктор

```
TreeSet()  
TreeSet(Collection <? extends E> c)  
TreeSet(Comparator<? Super E> компаратор)  
TreeSet( SortedSet<E> ss)
```

Отличие от HashSet в том что элементы расположены в порядке возрастания.

## Пример использования TreeSet

```
package com.gmail.tsa;
```

```
import java.util.*;
```

```
public class Main {
```

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

```
        TreeSet<Integer> ts=new TreeSet<Integer>();
```

```
        ts.add(34);  
        ts.add(3);  
        ts.add(102);  
        ts.add(-3);
```

```
        System.out.println(ts);
```

```
    }  
}
```

Создание класса TreeSet



Добавление элементов в набор



Результатом будет вывод отсортированного списка



# Интерфейс Queue

Описывает элемент данных очередь (первый вошел — первый вышел).

## Методы определенные в Queue

Метод	Описание
E element()	Возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, то передается исключение.
boolean offer(E obj)	Пытается добавить элемент в очередь. Вернет true, если элемент добавлен, и false, если наоборот.
E peek()	Вернет элемент из головы очереди. Элемент не удаляется. Вернет null, если очередь пуста.
E poll()	Вернет элемент из головы очереди. Элемент удаляется. Вернет null, если очередь пуста.
E remove()	Удаляет элемент из головы очереди, возвращая его.

# Интерфейс Deque

Описывает элемент данных очередь (первый вошел — первый вышел) или «стек» (последний вошел — первый вышел).

Методы	Описание
void addFirst(E obj)	Добавляет элемент в голову очереди.
void addLast(E obj)	Добавляет элемент в хвост очереди.
Iterator<E> descendingIterator()	Возвращает обратный итератор.
E getFirst()	Вернет первый элемент. Объект не удаляется.
E getLast()	Возвращает последний элемент. Объект не удаляется.
boolean offerFirst(E obj)	Пытается добавить объект в голову очереди.
boolean offerLast(E obj)	Пытается добавить объект в хвост очереди.
E peekFirst()	Возвращает элемент из головы очереди. Объект не удаляется.
E peekLast()	Возвращает элемент из хвоста очереди. Объект не удаляется.
E pollFirst()	Возвращает элемент из головы очереди. Объект удаляется.
E pollLast()	Возвращает элемент из хвоста очереди. Объект удаляется.
E pop()	Возвращает элемент из головы очереди. Объект удаляется.
void push(E obj)	Добавляет элемент в голову очереди.
E removeFirst()	Возвращает элемент из головы очереди. Объект удаляется.

## Класс ArrayDeque

Реализует интерфейс ArrayDeque

### Конструктор

```
ArrayDeque()  
ArrayDeque(int размер)  
ArrayDeque(Collection<? Extends E>c)
```

## Пример использования ArrayDeque

```
package com.gmail.tsa;
```

```
import java.util.*;
```

```
public class Main {
```

Создание очереди с элементами типа Integer

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

```
        ArrayDeque<Integer> ts=new ArrayDeque<Integer>();
```

```
        ts.push(34);  
        ts.push(3);  
        ts.push(102);  
        ts.push(-3);
```

Помещение элементов в очередь

```
        for(;ts.peek()!=null;){  
            System.out.println(ts.pop());  
        }
```

Выталкивание элементов из очереди

```
    }  
}
```



## Компараторы

Используется для указания порядка сортировки объектов в сортируемых коллекциях

Для этого требуется реализовать интерфейс `Comparator <T>`

Этот интерфейс должен реализовать два метода:

- `int compare (T obj_1, T obj_2)` — сравнение объектов

## Создание класса описывающего флешку

```
package com.gmail.tsa;

public class UsbDrive {
    int razm;
    String name;

    UsbDrive(int razm, String name){
        this.razm=razm;
        this.name=name;
    }

    public String toString(){
        return (name+" "+razm+" Gb");
    }
}
```

## Создание компаратора для класса UsbDrive

```
package com.gmail.tsa;
```

```
import java.util.Comparator;
```

```
public class UsbDriveComparator implements Comparator<UsbDrive>{
```

```
@Override
```

```
public int compare(UsbDrive a, UsbDrive b){
```

```
    if(a.razm>b.razm){  
        return -1;
```

```
    }
```

```
    else if(a.razm<b.razm){  
        return 1;
```

```
    }
```

```
    else return 0;
```

```
}
```

```
}
```

Класс реализует интерфейс Comparator



Переопределение метода compare

Этот метод должен вернуть положительное целое число, если первый параметр (в примере — a) больше второго (в примере — b), отрицательное число, если второй параметр больше первого и 0, если они равны.

# Использование Comparator

```
package com.gmail.tsa;

import java.util.*;

public class Main {

    public static void main(String[] args) {

        TreeSet<UsbDrive> ts=new TreeSet<UsbDrive>(new UsbDriveComparator());

        ts.add(new UsbDrive(8, "Kingstion"));
        ts.add(new UsbDrive(16, "Transdent"));
        ts.add(new UsbDrive(4, "Apach"));

        Iterator<UsbDrive> itr=ts.iterator();

        for(;itr.hasNext();){

            System.out.println(itr.next());

        }

    }
}
```

Компаратор как аргумент конструктора

Заполнение коллекции

Создание итератора

Получение элементов методами итератора

## Синхронизированные коллекции

Для создания синхронизированного экземпляра Collection можно использовать статические методы класса `java.util.Collections`:

- `synchronizedCollection(Collection obj)`
- `synchronizedList(List list)`
- `synchronizedMap(Map map)`
- `synchronizedSet(Set set)`
- `synchronizedSortedMap(SortedMap sortedMap)`
- `synchronizedSortedSet(SortedSet sortedSet)`

```
package com.gmail.tsa;
```

Создание синхронизированного списка

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

```
public class Main {
```

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

```
        List<Integer> list = Collections.synchronizedList(new ArrayList<Integer>());  
    }
```

Вызов статического метода класса `Collections` для получения синхронизированного списка

## Краткие итоги занятия

**Обобщения** — параметризованные типы, дают возможность создавать классы, методы и интерфейсы, способные оперировать объектами разных классов. На обобщения можно накладывать ряд ограничений, что может конкретизировать используемый тип данных и дает возможность использовать контроль за типами с помощью компилятора.

**Collection** — интерфейс описывающий классы контейнеры, т. е. классы, в которых хранятся экземпляры других классов.

Они описывают основные структуры данных, используемые в информатике:

- Списки;
- Очереди;
- Наборы;

Если структура данных поддерживает сортировку или располагает элементы данных в отсортированном порядке, то для этого нужно использовать экземпляр класса, реализующего интерфейс **Comparator**.

Можно создать синхронизированный объект Collection.

## Литература

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 563 — 590
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала — Java . Том 1. Основы. Девятое издание. стр. 683-720

## Домашнее задание

- 1) Написать метод, который создаст список, положит в него 10 элементов, затем удалит первые два и последний, а затем выведет результат на экран.
- 2) Модифицируйте класс «Группа» для более удобных методов работы с динамическими массивами.
- 3) Считайте из файла текст на английском языке, вычислите относительную частоту повторения каждой буквы и выведите на экран результат в порядке убывания относительной частоты повторения.
- 4) Шелдон, Леонард, Воловиц, Кутрапалли и Пенни стоят в очереди за «двойной колой». Как только человек выпьет такой колы, он раздваивается и оба становятся в конец очереди, чтобы выпить еще стаканчик. Напишите программу, которая выведет на экран состояние очереди в зависимости от того, сколько стаканов колы выдал аппарат с чудесным напитком. Например, если было выдано только два стакана, то очередь выглядит как:  
[Volovitc, Kutrapalli, Penny, Sheldon, Sheldon, Leonard, Leonard]



# Java OOP - Stream API

(Дополнительные материалы к лекции Generic and Collection)

Stream API - это одно из нововведений в Java 8.

Используя функциональные интерфейсы и лямбда-функции, Stream API предоставляет функциональный подход к обработке наборов данных.

`stream` — поток данных, порожденный структурой данных, который впоследствии может быть обработан с использованием функций высшего порядка.

**Составил: Цымбалюк А.Н.**

# Классификация stream

**stream**

Последовательные

Параллельные

Примитивные типы  
Более развиты числовые

Ссылочные типы  
Объекты произвольных классов

## Методы работы со stream

Промежуточные  
Принимают stream и  
порождают stream

Конечные  
Принимают stream и возвращают  
произвольный объект

**Stream API** - новый способ работать со структурами данных в функциональном стиле появившийся в Java 8.

Чаще всего с помощью stream в Java 8 работают с коллекциями, однако этот механизм может использоваться и для других структур данных.

Метод создания stream	Источник stream
Collection.stream()	Порождает поток на основе экземпляра коллекций
Stream.of(val1,val2,val3)	Порождает поток на основе набора данных разделенных запятыми
Arrays.stream( array )	Порождает поток на основе массива элементов
Files.lines(путь_к_файлу)	Порождает поток из файла (единицей выступит строка в файле)
Collection.parallelStream()	Порождает параллельный поток из экземпляра коллекций

## Промежуточные методы

Название	Аргумент	Описание
filter	Predicate<T>	Оставляет в потоке те элементы, для которых Predicate вернет true .
map	Function<T,R>	Переводит элементы потока в другой тип или выполняет преобразования.
sorted	Comparator<T,T>	Сортирует поток на основе компаратора.
flatMap flatMapToInt flatMapToDouble flatMapToLong	Function<T,R>	Позволяет упаковать несколько потоков в один.
mapToInt mapToDouble mapToLong	Function<T,примитив>	Методы для перевода примитивных типов.
limit	int size	Ограничивает поток size элементами.
skip	int size	Пропускает size элементов.

Все эти методы принимают поток и результатом их работы также будет поток

## Конечные методы

Название	Аргумент	Описание
collect	Supplier<T>	Сборка результата в элемент коллекции.
toArray	Supplier<T>	Сборка результат в массив.
reduce	BinaryOperator<T>	Операция накопления.
count		Считает количество элементов в потоке.
anyMatch	Predicate<T>	Вернет true, если в потоке есть хоть один элемент удовлетворяющий условию.
allMatch	Predicate<T>	Вернет true, если в потоке все элементы удовлетворяющий условию.
min	Comparator<T,T>	Вернет min на основе компаратора.
max	Comparator<T,T>	Вернет max на основе компаратора.
forEach	BinaryOperator<T>	Применит функцию к каждому элементу.
forEachOrdered	BinaryOperator<T>	Тоже самое, но с сохранением порядка.
findFirst		Вернет первый элемент из потока.

Все эти методы принимают поток и результатом их работа будет объект. Т.е. поток на таких методах прервется

## Predicate<T> и его использование в filter()

Predicate<T> - функциональный интерфейс появился в JDK 1.8. Переопределяется для того, чтобы указать, подходит объект по критерию или нет. Если подходит, то вернет true, если нет, то false. Метод для реализации - boolean test(T t). filter (Predicate<T> ) промежуточный метод, который оставит в потоке только те элементы, для которых реализация Predicate вернет true.

```
package com.gmail.tsa;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.stream.Collectors;
```

```
public class Main {
```

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

```
        ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 4, 5, 7, 8, 9));
```

```
        ArrayList<Integer> listOne = list.stream()
```

```
        .filter(n -> n % 2 == 0)
```

```
        .collect(Collectors.toCollection(ArrayList::new));
```

```
        System.out.println(listOne);
```

```
    }
```

```
}
```

Создание списка целых чисел



Порождение потока



Применение filter. Predicat реализован лямбдой



Результат в список



## Интерфейс Function<T,R> и использование в методах map()

Интерфейс Function<T,R> - появился в JDK 1.8 . Предназначен для создания объекта (произвольного в примере R — типа) на основе объекта другого типа (в примере T - типа). Метод для переопределения R apply(T t)

Например, когда нужно на основе списка чисел создать список строк, каждый элемент которого зависит (генерируется) от числа.

```
package com.gmail.tsa;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 4, 5, 7, 8, 9));

        ArrayList<String> listOne=list.stream()
        .map(n -> "Number "+n)
        .collect(Collectors.toCollection(ArrayList::new));

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

Применение map преобразуем (Integer → String)

## Comparator<T,T> и его использование в sort()

Функциональный интерфейс предназначенный для сравнения двух объектов одного типа. Чаще всего используется при сортировке. Метод для переопределения - `int compare(T o1, T o2)` - должен вернуть положительное число, если первый аргумент больше второго, отрицательное, если второй больше первого и 0, если они равны.

```
package com.gmail.tsa;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.stream.Collectors;
```

```
public class Main {
```

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

```
        ArrayList<Integer> list = new ArrayList<>(Arrays.asList(11, 4, 7, 5, 8, 9));
```

```
        ArrayList<Integer> listOne = list.stream()
```

```
        .sorted((a, b) -> a - b)
```

Применение sort.

```
        .collect(Collectors.toCollection(ArrayList::new));
```

Результат в список

```
        System.out.println(listOne);
```

```
    }
```

```
}
```

Создание списка целых чисел



Порождение потока





## Промежуточные методы можно объединять в цепочки

```
int[] array = { 2, 6, 3, 8, 1, 2, 3, 4, 5, -1, -2, 4 };
```

Порождение потока из массива

```
ArrayList<Integer> res = Arrays.stream(array)
```

```
.mapToObj(n -> n)
```

Перевод примитивного потока в ссылочный

```
.sorted((nOne, nTwo) -> Math.abs(nOne) - Math.abs(nTwo))
```

```
.collect(Collectors.toCollection(ArrayList::new));
```

Результат в список

Сортировка по величине модуля

## Примеры цепочки промежуточных методов

```
Map<Character, Long> ar = null;
try {
    ar = Files.lines(Paths.get("a.txt"))
        .map(n -> n.toUpperCase())
        .flatMapToInt(n -> n.chars())
        .filter(n -> n>='A' && n<='Z')
        .mapToObj((n) -> (Character.valueOf((char) n)))
        .collect(Collectors.groupingBy(n -> n, Collectors.counting()));
} catch (IOException e) {
    System.out.println(e);
}

for (Character key : ar.keySet()) {
    System.out.println(key + " - " + ar.get(key));
}
```

stream из файла

В верхний регистр

Каждую строку в отдельный поток

Фильтрация, что символ это буква

Преобразование

Сбор к карту символ — кол-во

## Коллекторы

Коллектор - конструкция общего характера для порождения составных значений из потоков. Коллектор можно использовать с произвольным потоком, передав его в качестве аргумента метода `collect`.

В стандартной реализации наиболее активно используют  
`util.stream.Collectors`

## Наиболее распространенные коллекторы

Коллекторы	Результат
toList	Список
toSet	Набор
toCollection	Любой элемент из коллекций
maxBy, minBy	Один элемент
groupBy	Карта

## Краткие итоги лекции

**Stream API** новое средство введенное в JDK 1.8 призванное частично реализовать функциональную парадигму программирования. В основе стоит идея потока данных, который можно породить из основных классов контейнеров. К потоку в дальнейшем можно применить промежуточные методы (только изменяют состояние потока, но не прерывают его) и конечные (по сути сбор результатов).

Большое количество промежуточных методов в качестве параметра используют функциональные интерфейсы такие, как Predicate, Function т. д.

Так как функциональные интерфейсы можно реализовать с помощью лямбда методов, это дает возможность писать короткий и лаконичный код.

Существует возможность сцепления промежуточных методов с целью преобразования потока данных.

Конечные методы призваны сохранить результат в какой-нибудь структуре или единственном элементе.

## Литература

Ричард Уорбэртон Лямбда выражения в Java 8. Функциональное программирование - в массы. Москва, 2014

## Домашнее задание

Внимание! Все приведенные ниже задания решить, используя stream API

1. Напишите методы, которые позволят выделить из группы студентов тех, у кого фамилия начинается с определенной буквы.
2. Напишите метод, который найдет в массиве целых чисел число, модуль которого ближе всего к 0. Если есть два таких числа (например 2 и -2), верните положительное.
3. На основе строки сгенерируйте массив целых чисел, где каждое число должно быть ASCII кодом соответствующей буквы.
4. Найдите максимальное число из набора чисел, которые хранятся в текстовом файле.

# Java OOP

## (Map)

Карта (map) — объект, сохраняющий ассоциации между ключами и значением (ключ-значение). По заданному ключу вы всегда можете найти значение. Ключи и значения могут быть любых (ссылочных) типов.



# Работа с картами (map)

Карта (map) — объект сохраняющий ассоциации между ключами и значением (ключ-значение). По заданному ключу вы всегда можете найти значение.

## Интерфейсы карт

Интерфейс	Описание
Map.Entry	Описывает элемент карты (ключ-значение) .
Map	Отображение ключа на значение.
SortedMap	Расширяет Map так, что ключи идут в порядке возрастания.
NavigableMap	Расширяет интерфейс SortedMap для обработки извлечения элементов на основе поиска по ближайшему соответствию.

**!** Карты не реализуют интерфейс Iterable

# Интерфейс Map.Entry

Позволяет работать с элементом карты — Map.Entry

```
interface Map.Entry<K,V>
```

Метод	Описание
boolean equals(Object obj)	Вернет true, если obj — это объект интерфейса Map.Entry, ключ и значение которого эквивалентны вызывающему объекту
K getKey()	Вернет ключ данного элемента карты
V getValue()	Вернет значение данного элемента карты
int hashCode()	Вернет хеш-код данного элемента карты
V setValue(V v)	Устанавливает значение данного элемента карты равным v

## Интерфейс Map

Интерфейс Map соотносит уникальные ключи со значениями.

Ключ <K> — объект, который используется для последующего извлечения данных.  
Значение <V>— объект, который хранится в карте.

```
interface Map<K,V>
```

# Методы определенные в интерфейсе Map

Метод	Описание
<code>void clear</code>	Удаляет все пары «ключ-значение» из карты.
<code>boolean containsKey(Object k)</code>	Проверяет, содержит ли карта ключ <code>k</code> .
<code>boolean containsValue(Object v)</code>	Проверяет, содержит ли карта значение <code>v</code> .
<code>Set&lt;Map.Entry&lt;K,V&gt;&gt;entrySet()</code>	Возвращает набор, содержащий все значения карты. Т.е. карта транслируется в набор.
<code>boolean equals (Object obj)</code>	Вернет <code>true</code> в случае, когда объект карты содержит одинаковые значения.
<code>V get (Object obj)</code>	Вернет значение ассоциированное с ключом.
<code>int hashCode()</code>	Вернет хеш-код.
<code>boolean isEmpty()</code>	Проверяет пуста ли карта.
<code>Set&lt;K&gt; keySet()</code>	Вернет набор ключей вызывающей карты.
<code>V put(K k, V v)</code>	Помещает элемент в карту, переписывая значению ассоциированное с ключом. Вернет <code>null</code> , если ключ до этого не существовал. В противном случае вернет предыдущее содержание.
<code>void putAll(Map&lt;? Extends K, ? extends V&gt; m)</code>	Помещает все значения из <code>m</code> в карту .
<code>V remove(Object k)</code>	Удаляет элемент, ключ которого равен <code>k</code> .
<code>int size()</code>	Возвращает количество пар в карте.
<code>Collection&lt;V&gt; values()</code>	Вернет коллекцию содержащую значения карты.

# Интерфейс SortedMap

Гарантирует размещение элементов в возрастающем порядке значений ключей

```
interface SortedMap<K,V>
```

Метод	Описание
Comparator<? Super K> comparator()	Возвращает компаратор карты. Если карта использует стандартный, то вернется null.
K firstKey()	Вернет первый ключ вызывающей карты.
SortedMap<K,V> headMap(K end)	Вернет отсортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше end.
K lastKey()	Возвращает последний ключ карты.
SortedMap<K,V> subMap(K start, K end)	Вернет карту, содержащую элементы вызывающей карты, ключ которых больше или равен start и меньше end.
SortedMap<K,V> subMap(K start)	Вернет карту, содержащую элементы вызывающей карты, ключ которых больше start.

# Интерфейс NavigableMap

Расширяет интерфейс SortedMap и определяет извлечение элементов на основе ближайшего соответствия ключу или ключам.

```
interface NavigableMap<K,V>
```

Метод	Описание
Map.Entry<K,V> ceilingEntry(K obj)	Выполняет поиск в карте наименьшего ключа $k \geq \text{obj}$ . Если найдет, то вернется значение, если нет то null.
K ceilingKey(K obj)	Выполняет поиск в карте наименьшего ключа $k \geq \text{obj}$ . Если найдет, то вернется ключ, если нет, то null.
NavigableSet<K> descendingKeySet()	Вернет объект интерфейса NavigableSet, содержащий ключи карты в обратном порядке
NavigableMap<K,V> descendingMap()	Вернет объект интерфейса NavigableSet обратный вызывающей карте.
Map.Entry<K,V> firstEntry()	Вернет первое вхождение на карте.
Map.Entry<K,V> floorEntry(k obj)	Ищет на карте наибольшего ключа $k \leq \text{obj}$ . Если найдет, то вернется значение, если нет, то null.
K floorKey(k obj)	Выполняет поиск в карте наибольшего ключа $k \leq \text{obj}$ . Если найдет, то вернется ключ, если нет, то null.
NavigableMap<K,V> headMap(K top, boolean on)	Возвращает объект интерфейса NavigableSet , включающей все вхождения вызывающей карты, имеющей ключ меньше top.

# NavigableMap методы (продолжение)

Метод	Описание
Map.Entry<K,V> higherEntry(k obj)	Выполнит поиск в наборе наибольшего ключа, k> obj, и, если ключ найден вернется значение, а если нет, то null.
K higherKey(k obj)	Выполнит поиск в наборе наибольшего ключа, k< obj. Если ключ найден вернется ключ, если нет, то null.
Map.Entry<K,V> lastEntry()	Вернет последнее вхождение на карте. Т.е. значение с самым большим ключом.
Map.Entry<K,V> lowerEntry(k obj)	Выполнит поиск в наборе наибольшего ключа, k< obj. Если ключ найден, вернется значение, если нет, то null.
K lowerKey(K obj)	Выполнит поиск в наборе наибольшего ключа, k< obj. Если ключ найден, вернется ключ, если нет, то null.
NavigableSet<K> navigableKeySet()	Вернется объект интерфейса NavigableSet, содержащий ключи вызывающей карты. При пустой карте вернется null.
Map.Entry<K,V> pollFirstEntry()	Возвращает первое значение, при этом удаляя его.
NavigableMap<K,V> subMap(k low, boolean onlow, k top, boolean ontop)	Вернется объект интерфейса NavigableSet, содержащий ключи вызывающей карты, принадлежащие [low,top].
NavigableMap<K,V> tailMap(k low, boolean onlow)	Вернется объект интерфейса NavigableSet, содержащий ключи вызывающей карты, которые больше top.

# Классы реализующие интерфейс Map

Класс	Описание
AbstractMap	Реализует большую часть интерфейса Map.
EnumMap	Расширяет класс AbstractMap для использования с ключами типа enum (перечисления).
HashMap	Расширяет класс AbstractMap для использования хеш-таблицы.
TreeMap	Расширяет класс AbstractMap для использования дерева.
WeakHashMap	Расширяет класс AbstractMap для использования хеш-таблицы со слабыми ключами.
LinkedHashMap	Расширяет интерфейс HashMap, разрешая перебор в порядке вставки.
IdentityHashMap	Расширяет класс AbstractMap и использует проверку ссылочной эквивалентности при сравнении документов.



## Класс HashMap

Расширяет класс AbstractMap и реализует интерфейс Map. Использует хеш-таблицу для хранения карты, что обеспечивает постоянное время выполнения методов put(), get() в независимости от объема данных.

### Объявление:

```
class HashMap<K,V>
```

K-тип ключей

V-тип значений

### Конструкторы:

```
HashMap()
```

```
HashMap(Map<? Extends K, ? extends V> m)
```

```
HashMap(int емкость)
```

```
HashMap(int емкость, float коэф. заполнения)
```

## Пример использования HashMap

Вспомогательный класс Student предназначенный для помещения в карту

```
package com.gmail.tsa;

public class Student {

    String FIO;
    String facultet;
    int course;
    double oc;
    Student(String FIO, String facultet, int course, double oc){
        this.FIO=FIO;
        this.facultet=facultet;
        this.course=course;
        this.oc=oc;
    }

    public String toString(){
        return ("Student: "+FIO+" facultet "+facultet+" curs "+course);
    }
}
```

## Пример использования HashMap

```
package com.gmail.tsa;  
import java.util.*;
```

```
public class Main {
```

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

Создание объекта класса HashMap

```
        HashMap<String, Student> hm=new HashMap<>();
```

```
        hm.put("Petrov", new Student("Petrov Petr Petrovich", "physics", 2, 9.8));  
        hm.put("Sidorov", new Student("Sidorov Sidr Sidorovich", "physics", 1, 10.0));  
        hm.put("Ivanov", new Student("Ivanov Ivan Ivanovich", "geograsy", 5, 7.0));
```

```
        System.out.println(hm.get("Petrov"));
```

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

Заполнение карты парами  
ключ-значение

```
        Set<Map.Entry<String, Student>> hms=hm.entrySet();
```

```
        for (Map.Entry<String, Student> hmse:hms){
```

```
            System.out.println(hmse.getKey()+"\t"+hmse.getValue());
```

Получение ключа и значения пары

Объявление и получение  
набора из карты

```
        }
```

```
    }
```

```
}
```

Для набора Set можно объявить итератор

```
Iterator<Map.Entry<String, Student>> itr=hm.entrySet().iterator();  
for(;itr.hasNext();){  
    System.out.println(itr.next());  
}
```

Также можно получить только набор ключей

```
Set<String> set= hm.keySet();  
for(String str:set){  
    System.out.println(str);  
}
```

## Изменение значения по ключу

```
Student a=hm.get("Ivanov");
```

Получаем объект по ключу

```
a.course=4;
```

Изменяем его свойство

```
hm.put("Ivanov",a);
```

Заменяем значение на ключе

```
System.out.println(hm.get("Ivanov"));
```

Убедимся, что изменение произошло

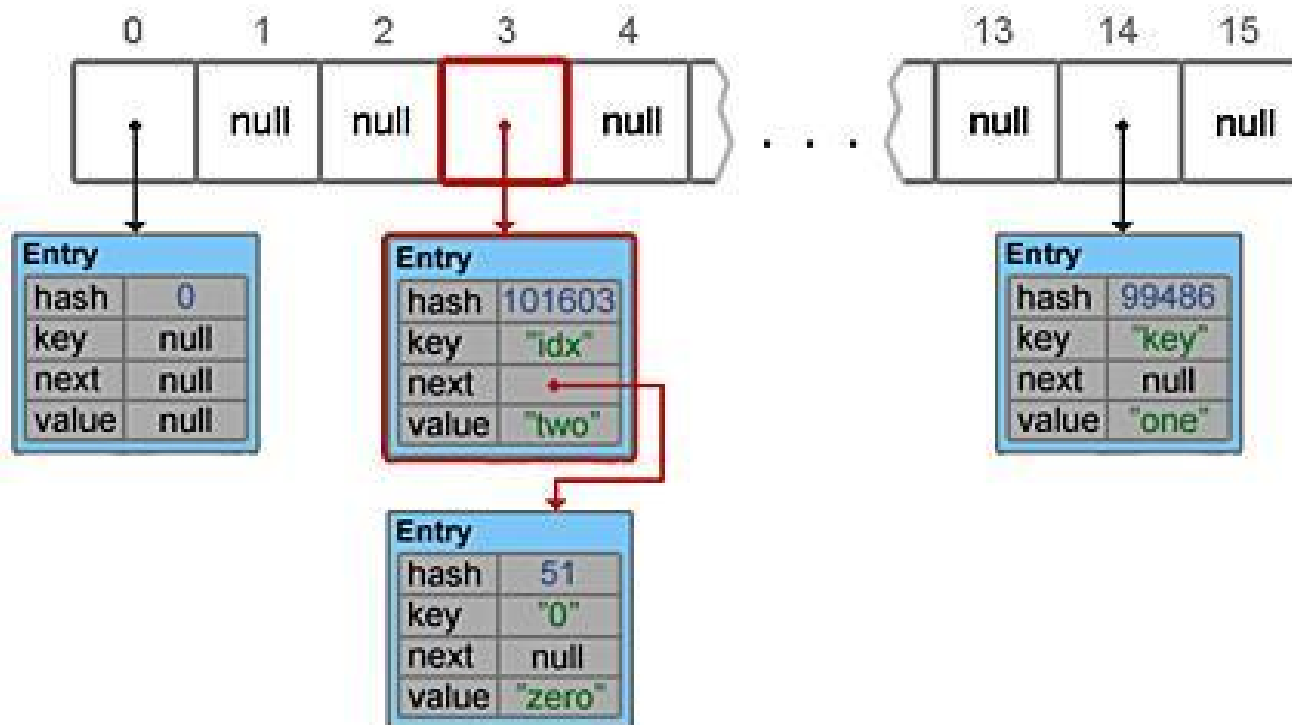
Т.е. при записи значения по уже существующему ключу, предыдущее значение затирается

# Как HashMap устроен внутри?

HashMap внутри - массив связанных списков. Элемент связанного списка - объект класса Entry, содержит ключ, значение и ссылку на следующий Entry.

Как искать элемент?

Поиск производится по ключу. По hashCode() ключа находим нужный бакет. Нашли бакет - значит нашли нужный нам связанный список. Далее проходим по списку и сравниваем ключи элементов списка с нашим ключом для поиска. Сравнение производим по функции equals(). Вернула функция true - значит нашли.



# Класс TreeMap

Расширяет класс AbstractMap и реализует интерфейс NavigableMap. Он создает карту, которая размещена в древовидной структуре. И карта-дерево гарантированно отсортирует элементы в порядке возрастания ключа.

Класс TreeMap является обобщенным классом

Объявление:

TreeMap()

TreeMap(Comparator<? Super K> компаратор)

TreeMap(Map<? Extends K, extends V> m)

TreeMap(SortedMap<K,? extends V> sm)

## Пример использования класса TreeMap

```
package com.gmail.tsa;  
import java.util.*;
```

```
public class Main {
```

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

```
        TreeMap<String, Student> tm=new TreeMap<>();
```

Объявление карты

```
        tm.put("Petrov", new Student("Petrov Petr Petrovich", "physics", 2, 9.8));  
        tm.put("Sidorov", new Student("Sidorov Sidr Sidorovich", "physics", 1, 10.0));  
        tm.put("Ivanov", new Student("Ivanov Ivan Ivanovich", "geogragy", 5, 7.0));
```

```
        Set<Map.Entry<String, Student>> set=tm.entrySet();
```

```
        for(Map.Entry<String, Student> s:set){  
            System.out.println(s.getKey()+" : "+s.getValue());  
        }
```

Заполнение карты

```
        System.out.println("Search student");
```

```
        Map.Entry<String, Student> s1=tm.ceilingEntry("Iv");  
        System.out.println(s1.getKey()+" : "+s1.getValue());
```

Поиск по частичному совпадению ключа



## Компараторы

Используется для указания порядка сортировки объектов в сортируемых картах.  
Сортировка производится по ключам.

Для этого требуется реализовать интерфейс `Comparator <T>`

Этот интерфейс должен реализовать метод:

- `int compare (T obj_1, T obj_2)` — сравнение объектов

# Алгоритмы коллекций

Инфраструктура коллекций определяет несколько методов, которые реализуют базовые алгоритмы обработки данных. Эти алгоритмы определены как статические методы в классе Collections.


## Некоторые методы из класса Collections

Метод	Описание
<code>boolean disjoint(Collection &lt;?&gt; a, Collection &lt;?&gt; b)</code>	Сравнивает элементы в a, с элементами в b. Вернет true, если коллекции не содержат одинаковых элементов.
<code>int frequency(Collection &lt;?&gt; a, Object obj)</code>	Подсчитывает, сколько раз элемент obj встречается в коллекции «a».
<code>void shuffle(List&lt;T&gt; список)</code>	Перемешивает случайным образом список
<code>&lt;T extends Object &amp; Comparable &lt;? super T&gt;&gt; T max(Collection&lt;? extends T&gt; c)</code>	Возвращает максимальный элемент из c
<code>&lt;T extends Object &amp; Comparable &lt;? super T&gt;&gt; T min(Collection&lt;? extends T&gt; c)</code>	Вернет минимальный элемент из c


## Пример использования Collections.Algorithms

```
package com.gmail.tsa;  
import java.util.*;  
public class Main {  
  
    public static void main(String[] args) {  
  
        ArrayList<Integer> a=new ArrayList<Integer>();  
        ArrayList<Integer> b=new ArrayList<Integer>();  
  
        a.add(12);  
        a.add(13);  
        a.add(45);  
  
        b.add(11);  
        b.add(2);  
        b.add(5);  
  
        System.out.println(Collections.disjoint(a, b));  
  
    }  
}
```

Заполнение двух множеств



Метод сравнивающий  
множества на пересечение



## Краткие итоги урока

Карта (**Map**) — интерфейс, указывающий на однозначное отображение уникального ключа, на значение.

**Map** — не является элементом Collection. По этой причине не реализует интерфейс Iterator, а, следовательно, по карте нельзя пройти, перебирая элементы.

Существует три основных интерфейса Map:

- 1) **Map** — просто отображение ключа на значение;
- 2) **SortedMap** — ключи хранятся в порядке возрастания;
- 3) **NavigableMap** — поиск по частичному совпадению;

Наиболее популярной реализацией карт является HashMap. В основе этой карты лежит механизм работы с хеш-таблицами, что уменьшает время доступа к элементу в случае больших объемов данных.

## Литература

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 600 — 617
- Кей Хорстманн, Гари Корнелл. Библиотека профессионала — Java . Том 1. Основы. Девятое издание. стр. 716-725

## Домашнее задание

1. Написать программу - переводчик, которая будет переводить текст в файле English.in, написанный на английском языке, на украинский язык, согласно заранее составленному словарю. Результат сохранить в файл Ukrainian.out.
2. Сделать ф-ю ручного наполнения словаря и возможность его сохранения на диск.
3. Решить задачу подсчета повторяющихся элементов в массиве с помощью HashMap.
4. Реализуйте программу, которая сопоставит каждой букве ее представление в виде ASCII - art, например

A →

```

      *
    * *
  *   *
*****
  *   *
    * *
```

Ваша программа должна дать возможность вывода произвольного текста на экран в виде его ASCII-art представления.

# Java OOP — Map and Optional в Java 8

(Дополнительные материалы к лекции Map)

Java 8 добавила к интерфейсу Map ряд методов для более удобной работы. Также был добавлен Optional как средство для уменьшения проблем при возникновении null ссылки.

## Новые методы в Map

В Java 8 было добавлено несколько новых методов для работы с Map:

- `getOrDefault()`
- `forEach()`
- `replaceAll()`
- `putIfAbsent()`
- `remove(Object key, Object value)`
- `computeIfAbsent()`
- `computeIfPresent()`
- `merge()`



# Метод `getOrDefault(k key, V defaultValue)`

Метод для получения элемента карты. Если по ключу `key` есть объект, то он будет возвращен. Если же такого ключа нет, то будет возвращено значение по умолчанию.

```
package com.gmail.tsa;
```

```
import java.util.HashMap;
```

```
public class Main {
```

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

```
        HashMap<Integer, String> map = new HashMap<>();
```

```
        map.put(1, "one");  
        map.put(2, "two");
```

Добавление элементов в карту

```
        String resultOne = map.getOrDefault(1, "DefaultValue");
```

```
        String resultTwo = map.getOrDefault(5, "DefaultValue");
```

```
        System.out.println(resultOne);  
        System.out.println(resultTwo);
```

```
    }
```

```
}
```

Получение элемента по ключу: ключ 1 есть в карте, поэтому вернется значение «one», а вот ключа 5 в карте нет, поэтому вернется значение по умолчанию «DefaultValue» .

Создание HashMap (реализует Map)



# Метод `forEach(BiConsumer<K key, V value>)`

Метод `forEach` — предназначен для выполнения действия для каждого элемента карты. Параметром этого метода является функциональный интерфейс `BiConsumer`, который принимает два значения на вход, и выполняет над ними действие не возвращающее результат.

```
package com.gmail.tsa;
```

```
import java.util.HashMap;
```

```
public class Main {
```

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

```
        HashMap<Integer, String> map = new HashMap<>();
```

```
        map.put(1, "one");  
        map.put(2, "two");  
        map.put(3, "three");
```

Добавление элементов в карту

```
        map.forEach((key, value) -> System.out.println(key + " - " + value));
```

```
    }
```

Вызов метода `forEach`. В примере интерфейс `BiConsumer` реализован с помощью лямбда функции. Т.е. выведет на экран значение ключа и значения карты.

Создание `HashMap` (реализует `Map`)

## Метод replaceAll(BiFunction<? super K,? super V,? extends V> function)

Метод для изменения значений по всем ключам карты. Параметром метода выступает функциональный интерфейс BiFunction(K,V,V). Первые два параметра - это входные аргументы этой функции. Последний элемент - это тип возвращаемого значения.

```
package com.gmail.tsa;
```

```
import java.util.HashMap;
```

```
public class Main {
```

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

```
        HashMap<Integer, String> map = new HashMap<>();
```

```
        map.put(1, "one");  
        map.put(2, "two");  
        map.put(3, "three");
```

Добавление элементов в карту

```
        map.replaceAll((key, value) -> "Number " + value);
```

```
        map.forEach((key, value) -> System.out.println(key + " - " + value));
```

```
    }
```

```
}
```

Создание HashMap (реализует Map)



Вызов метода replaceAll. Параметрами являются ключ и его значение. Этот метод изменит значение на результат работы этого метода. В примере реализовано лямбда функцией т. е., к каждому значению добавляется слово «Number».

## Метод putIfAbsent(K key, V value)

Метод применяется для установки значения по ключу (key). Если этого ключа не было в карте или он был ассоциирован с null, то элемент добавится в карту. Если же по указанному ключу было значение, то добавления не произойдет.

```
package com.gmail.tsa;

import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        HashMap<Integer, String> map = new HashMap<>();

        map.put(1, "one");
        map.put(2, "two");
        map.put(3, "three");

        map.putIfAbsent(3, "new three");
        map.putIfAbsent(4, "four");

        map.forEach((key, value) -> System.out.println(key + " - " + value));
    }
}
```

Попытка добавления на ключ **3 значения «new three»**. Неудачная, т. к. ключ 3 ассоциирован с нормальным значением. Попытка добавить на ключ **4 значение «four»** удачная так, как такого ключа в карте еще не было.

## Метод remove(Object key, Object value)

Метод удалит ключ (key) только в том случае, если он отображается на указанное значение. В противном случае удаления не происходит.

```
package com.gmail.tsa;

import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        HashMap<Integer, String> map = new HashMap<>();

        map.put(1, "one");
        map.put(2, "two");
        map.put(3, "three");

        map.remove(3, "new three");

        map.remove(2, "two");

        map.forEach((key, value) -> System.out.println(key + " - " + value));
    }
}
```

Попытка удаления элемента отображения. Неудачная, так как 3 не отображается на «new three».

Попытка удаления элемента отображения. Удачная, так как 2 действительно отображается на «two».

## Метод computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)

Метод для вычисления значения на основании ключа при условии, что такого ключа не было или он отображался на null. Второй параметр - это функциональный интерфейс Function метод, которого вычисляет значение на основании ключа.

```
package com.gmail.tsa;

import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        HashMap<Integer, String> map = new HashMap<>();

        map.put(1, "one");
        map.put(2, "two");
        map.put(3, "three");

        map.computeIfAbsent(3, key -> String.valueOf(key));

        map.computeIfAbsent(4, key -> String.valueOf(key));

        map.forEach((key, value) -> System.out.println(key + " - " + value));
    }
}
```

Попытка вычисления значения для ключа 3. Неудачная, так как ключ 3 имеет значение.

Попытка вычисления значения для ключа 4. Удачная, так как ключ 4 не имеет значение. Для вычисления использовалась лямбда функция.

Метод `computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`

Метод используется для изменения значения по существующему ключу, если значение существовало, то новое значение будет вычислено на основе старого и помещено в карту. Однако если, новое значение равно `null`, то этот ключ будет удален из карты.

```
package com.gmail.tsa;

import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        HashMap<Integer, String> map = new HashMap<>();

        map.put(1, "one");
        map.put(2, "two");
        map.put(3, "three");

        map.computeIfPresent(3, (key, value) -> value + " number");

        map.computeIfPresent(4, (key, value) -> value + " number");

        map.forEach((key, value) -> System.out.println(key + " - " + value));

    }
}
```

Изменение значения на ключе 3. Удачное, так как 3 есть в карте. Для ключа 4 будет неудачно.



Метод merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)

Метод получит значение по указанному ключу (key). Если оно null, то на ключ запишется значение по умолчанию (value). Если не null, то определится новое значение ключа (на основе remappingFunction) и заменит старое на указанном ключе.

```
package com.gmail.tsa;
```

```
import java.util.HashMap;
```

```
public class Main {
```

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

```
        HashMap<Integer, String> map = new HashMap<>();
```

```
        map.put(1, "one");
```

```
        map.put(2, "two");
```

```
        map.put(3, "three");
```

```
        map.merge(3, "3", (key, value) -> value + " number");
```


```
        map.merge(4, "four", (key, value) -> value + " number");
```

```
        map.forEach((key, value) -> System.out.println(key + " - " + value));
```

```
    }
```

```
}
```

Попытка изменения на ключе 3. Так как на ключе есть значение, оно определится на основе функции. Для 4 возьмется значение по умолчанию.





## Класс Optional

В JDK 1.8 был введен дополнительный класс Optional. Это обобщенный класс контейнер, который может содержать или не содержать null. Позволяет существенно уменьшить количество проверок на null.

Используя Optional, можно заранее узнать параметром является null или нормальное значение

Метод	Описание
<code>static &lt;T&gt; Optional &lt;T&gt; ofNullable(T val)</code>	Создается Optional с заданным value. Если value==null, то вернется пустой Optional.
<code>T orElse(T val)</code>	Если значение было, то вернется значение, в противном случае val.
<code>T orElseGet(Supplier&lt;? Extends T&gt; func)</code>	Если значение было вернется оно, если не было, вернется сгенерированное с помощью func значение.

На этом и следующем слайде приведен список методов класса Optional

## Методы Optional

Метод	Описание
<code>static &lt;T&gt; Optional&lt;T&gt; empty()</code>	Возвращает пустой объект класса Optional.
<code>boolean equals (Object obj)</code>	Сравнение двух объектов.
<code>Optional &lt;T&gt; filter (Predicate&lt;? super T&gt; pr)</code>	Возвращает Optional со свойством как и у вызывающего объекта, при условии что это значение в предикате даст true.
<code>U Optional &lt;U&gt; flatMap(Function&lt;? super T, Optional&lt;U&gt; &gt; func)</code>	Применяет func к вызывающему объекту, если объект содержит значение. Если нет, то пустой объект.
<code>T get()</code>	Возвращает значение. Если значение пустое генерируется исключение.
<code>int hashCode()</code>	Вернет хеш-код объекта.
<code>void ifPresent(Consumer&lt;? super T&gt; func)</code>	Вызывает func, если объект не пуст. В противном случае ничего не происходит.
<code>boolean isPresent()</code>	Если объект не пуст, то true, false если пуст.
<code>U Optional &lt;U&gt; Map(Function&lt;? super T, ? extends U &gt; func)</code>	Применяет func к вызывающему объекту, если значение было то вернется результат иначе пустой объект.
<code>Static &lt;T&gt; Optional &lt;T&gt; of (T value)</code>	Создает Optional с указанным value.

# Примеры использования Optional

```
package com.gmail.tsa;
```

```
import java.util.Optional;
```

```
public class Main {
```

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

```
        Optional<String> a = Optional.ofNullable(null);
```

```
        print(a);
```

```
        a = Optional.ofNullable("Hello world");
```

```
        print(a);
```

```
    }
```

```
    public static void print(Optional<String> text) {
```

```
        System.out.println(text.orElse("Nothing"));
```

```
    }
```

```
}
```

Создание объекта Optional, если параметр null то объект будет пустой

Теперь Optional не пуст, и метод выведет на экран его содержимое.

На экран выведется содержимое Optional, или если этот объект пуст, то слово Nothing

## Пример использования Optional

```
package com.gmail.tsa;
```

```
import java.util.Optional;
```

```
public class Main {
```

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

```
        Optional<Integer> a = Optional.ofNullable(null);
```


```
        Integer b = a.orElse(10);
```

```
        System.out.println(b);
```


```
    }
```

```
}
```

Создание объекта Optional: если параметр null, то объект будет пустой



Получения значения из объекта Optional. Если объект не пуст, то вернется значение. Если пуст, то вернется параметр метода orElse(). В данном примере 10.



# Пример использования Optional в StreamAPI

Результатом многих конечных методов в StreamAPI являются переменные типа Optional.

```
package com.gmail.tsa;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Optional;

public class Main {

    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<>(Arrays.asList( 1, 2, 9, 3, 5, 10));

        Optional<Integer> result = list.stream()
            .filter(n -> n%2==0)
            .max((a,b) -> a-b);

        System.out.println(result.get());
    }
}
```

Создание списка из массива

Optional — результат работы метода max()

Метод get для получения результата

## Краткие итоги лекции

В JDK 1.8 добавлено много методов для работы с Map. Среди них есть и те, что используют функциональные интерфейсы, и, как следствие, могут быть реализованы с помощью лямбда функций.

Optional — новый класс, введенный в JDK 1.8, для уменьшения количества кода, связанного с проверкой на null.

## Литература

- <https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>
- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 656 - 659

# Java OOP

## (Сетевое программирование)

Сокет — абстрактное понятие, поддерживаемое программным обеспечением для организации операции обмена данными по сети.



## Класс InetAddress

Класс используется для хранения числового значения IP адреса или его доменного имени. Может работать как с IP4, так и с IP6.

Класс не имеет явных конструкторов. Для создания используются методы фабрики (статические методы класса возвращают экземпляр класса)

`static InetAddress getLocalHost ()` - по локальному хосту

`static InetAddress getByName (String имя хоста)` — по локальному хосту

`static InetAddress[] getAllByName(String имя хоста)`

`static InetAddress getByAddress()` - по IP адресу

**!**Классы для работы с сетью хранятся в пакете `java.net`

## Методы класса InetAddress

Метод	Описание
<code>boolean equals(Object add)</code>	Сравнивает два адреса в интернете.
<code>byte[] getAddress()</code>	Вернет байтовый массив, представляющий IP.
<code>String getHostAddress()</code>	Вернет строку, представляющей адрес хоста, ассоциированного с объектом класса <code>InetAddress</code> .
<code>String getHostName()</code>	Вернет строку, представляющей имя хоста, ассоциированного с объектом класса <code>InetAddress</code> .
<code>boolean isMulticastAddress()</code>	Проверяет является ли адрес групповым.
<code>String toString()</code>	Вернет строковое представление.

## Пример создания InetAddress

```
package com.gmail.tsa;
```

```
import java.net.*;
```

← Импорт необходимого пакета

```
public class Main {
```

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

Создание нового объекта

```
        try{
```

```
            InetAddress adress=InetAddress.getByName("www.google.com");
```

```
            System.out.println(adress);
```

```
        } catch(UnknownHostException e){
```

```
            System.out.println(e.toString());
```

```
        }
```

↑ Вывод IP на экран

```
    }
```

```
}
```

## Класс URL, URLConnection

### Класс **URL**

Предназначен для поиска содержимого в сети по его адресу:

Конструктор:

URL(String спецификатор)

URL(String имя протокола, String имя хоста, int порт, String путь)

URL(String имя протокола, String имя хоста, String путь)

URL(URL obj, String спецификатор)

### Класс **URLConnection**

Предназначен для доступа к содержимому в сети :

Объект этого класса создается с помощью метода `openConnection()` объекта класса URL

## Методы класса URLConnection

Метод	Описание
<code>int getLength()</code>	Возвращает размер в байтах содержимого связанного с ресурсом. Если ресурс недоступен вернет -1.
<code>long getLengthLong()</code>	Возвращает размер в байтах содержимого связанного с ресурсом. Если ресурс недоступен вернет -1.
<code>String getContentType()</code>	Вернет тип содержимого ресурса. (Указан в content-type).
<code>long getDate()</code>	Вернет время и дату ответа (в мсек с 1970 года 1 января) .
<code>long getExpiration()</code>	Вернет дату устаревания ресурса.
<code>InputStream getInputStream()</code>	Вернет объект класса InputStream связанный с ресурсом.
<code>String getHeaderField(int index)</code>	Вернет значение заголовочного поля по индексу.
<code>String getHeaderField(String fieldname)</code>	Вернет значение заголовочного поля по имени.
<code>String getHeaderFieldKey(int index)</code>	Вернет ключ заголовочного поля по индексу.
<code>Map&lt;String,List&lt;String&gt;&gt; getHeaderFields()</code>	Вернет карту всех заголовочных полей и их значений.
<code>long getLastModified()</code>	Вернет время и дату последнего изменения ресурса.

## Пример использования класса URL

```
package com.gmail.tsa;

import java.net.*;

public class Main {

    public static void main(String[] args) {


        try{
            URL url=new URL("http://ya.ru/");

            System.out.println(url.getProtocol());
            System.out.println(url.getHost());
            System.out.println(url.getPort());
            System.out.println(url.getFile());
            System.out.println(url.getUserInfo());
        }
        catch(MalformedURLException e){
            System.out.println(e.toString());
        }

    }

}
```

Создание объекта URL



Использование методов класса URL



## Пример использования URLConnection

```
package com.gmail.tsa;


import java.net.*;
import java.io.*;

public class Main {


    public static void main(String[] args) {

        try{
            int c;
            URL url=new URL("https://www.google.ru/?gws_rd=ssl");
            URLConnection urlc=url.openConnection();
            System.out.println(urlc.getDate());
            System.out.println(urlc.getContentType());
            long l=urlc.getContentLengthLong();
            if(l!=0){
                System.out.println();
                InputStream ins=urlc.getInputStream();
                for(;(c=ins.read())!=-1;){
                    System.out.print((char)c);
                }
            }
        }
        catch(Exception e){
            System.out.println(e.toString());
        }
    }
}
```

Создание объекта URL



Создание объекта URLConnection  
и демонстрация его методов



Вывод содержимого ответа в консоль



## Класс HttpURLConnection

Является подклассом `URLConnection` и обеспечивает поддержку HTTP соединений. Создается этот класс также с помощью метода `openConnection()` объекта класса `URL`. Для этого используется преобразование типов.

```
HttpConnection hpc=(HttpConnection) url.openConnection();
```



## Некоторые методы класса HttpURLConnection

Методы	Описание
<code>static boolean getFollowRedirects()</code>	Вернет true, если осуществляется перенаправление.
<code>String getRequestMethod()</code>	Вернет строковое представление метода выполнения запроса.
<code>int getResponseCode()</code>	Вернет код ответа http. Если код ответа не может быть получен, то вернется -1.
<code>String getResponseMessage()</code>	Вернет сообщение ответа ассоциированное с кодом ответа.
<code>static void setFollowRedirects( boolean)</code>	Позволяет установить автоматическое (true) или нет (false) перенаправление.
<code>void setRequestMethod(String)</code>	Устанавливает метод, которым выполняются запросы. По умолчанию это метод GET, но доступны и методы POST.

# Пример использования HttpURLConnection

```
package com.gmail.tsa;

import java.net.*;
import java.io.*;
import java.util.*;

public class Main {

    public static void main(String[] args) {
        try{
            URL url=new URL("https://www.google.ru/?gws_rd=ssl");
            HttpURLConnection urlc=(HttpURLConnection) url.openConnection();

            System.out.println("Метод запроса на сервер "+urlc.getRequestMethod());
            System.out.println("Тип ответа "+urlc.getResponseMessage());
            Map<String,List<String>> hm=urlc.getHeaderFields();
            Set<String> hdrf=hm.keySet();
            for(String k:hdrf){
                System.out.println("Key = "+k+" : "+ "Value = "+hm.get(k));
            }
        }
        catch(Exception e){
            System.out.println(e.toString());
        }
    }
}
```

Создание HttpURLConnection

Создание нового объекта URL

## Клиентские сокеты TCP/IP

Сокеты TCP/IP предназначены для реализации надежных двунаправленных, постоянных соединений между точками (хостами) в сети на основе потоков.

В java реализованы два вида сокетов TCP для клиентов и для серверов.  
ServerSocket - «слушатель» предназначен для серверов.  
Socket - предназначен для клиентов.

Конструкторы для клиентского сокета:

Socket(String имя хоста, int порт)  
Socket(InetAddress ip адрес, int номер порта)

## Методы класса Socket

Методы	Описание
InetAddress getInetAddress()	Вернет объект класса InetAddress, ассоциированный с объектом класса Socket . В случае если сокет не подключен, вернется null.
int getPort()	Возвращает удаленный порт, к которому подключен вызывающий объект класса Socket. Если сокет не подключен, возвращается 0.
int getLocalPort()	Возвращает локальный порт, к которому привязан вызывающий объект класса Socket. Если сокет не подключен, возвращается -1.
InputStream getInputStream()	Возвращает объект класса InputStream ассоциированный с вызывающим сокетом.
OutputStream getOutputStream()	Возвращает объект класса OutputStream ассоциированный с вызывающим сокетом.

**!Объект Socket должен быть закрыт методом close()**

## Пример использования класса Socket

```
package com.gmail.tsa;
```

```
import java.net.*;
```

```
import java.io.*;
```

```
public class Main {
```

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

```
        try(Socket socket = new Socket("ya.ru", 80)){
```

Создание клиентского сокета

```
            PrintWriter pw=new PrintWriter(socket.getOutputStream());
```

```
            pw.println("GET / HTTP/1.1");  
            pw.println("Host:ya.ru");  
            pw.println("");  
            pw.flush();
```

Запись в выходной поток запроса

```
            InputStream s = socket.getInputStream();  
            int r;  
            for(;;(r=s.read())!=-1){  
                System.out.print((char)r);  
            }
```

Открытие выходного потока и вывод  
ответа в консоль

```
        }  
        catch(Exception e){  
            System.out.println(e.toString());  
        }
```

```
    }
```

```
}
```

## Серверный сокет. ServerSocket

**ServerSocket** используются для создания серверов, которые прослушивают обращения локальных или удаленных программ, которые устанавливают соединения с ним через открытые порты.

Конструкторы:

`ServerSocket(int порт)`

`ServerSocket(int порт, int максимальная очередь)`

`ServerSocket(int порт, int максимальная очередь, InetAddress локальный адрес)`

Метод **accept()** реализует блокирующий вызов, который будет ожидать от клиента инициализации соединения, а затем вернет объект класса `Socket`, который и будет взаимодействовать с клиентом.

# Пример использования ServerSocket

```
package com.gmail.tsa;
```

```
import java.util.*;
```

```
import java.io.*;
```

```
import java.net.*;
```

```
public class Server {
```

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

```
        try{
```

```
            ServerSocket s=new ServerSocket(20000);
```

```
            Socket incoming=s.accept();
```

```
            try{
```

```
                Scanner sc=new Scanner(incoming.getInputStream());
```

```
                PrintWriter out=new PrintWriter(incoming.getOutputStream(),true);
```

```
                boolean exit=true;
```

```
                for(;exit;){
```

```
                    String line=sc.nextLine();
```

```
                    out.println("Ответ сервера "+line);
```

```
                    if(line.compareTo("exit")==0) exit=false;
```

```
                }
```

```
            }
```

```
            finally{
```

```
                incoming.close();
```

```
            }
```

```
        }
```

```
        catch(IOException e){
```

```
            e.printStackTrace();
```

```
        }
```

```
        System.out.println("END");
```

```
    }
```

```
}
```

Создание нового серверного сокета

Получение входных и выходных потоков

Получение запросов пользователя и генерация ответов

## Программа telnet

TELNET (англ. TErminaL NETwork) — сетевой протокол для реализации текстового интерфейса по сети (в современной форме — при помощи транспорта TCP). Название «telnet» имеют также некоторые утилиты, реализующие клиентскую часть протокола.

Для тестирования необходимо запустить telnet 127.0.0.1 20000



## Простой HttpServer на Socket

Цель работы — написать простой сервер, который, при обращении к нему, будет отдавать html страничку со списком студентов.

## Класс Client

```
package com.gmail.tsa;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
```

```
public class Client implements Runnable {
```

```
    private Socket soc;
    private String ansv;
    private Thread t;
```

Входящий сокет обеспечит ServerSocket

```
    public Client(Socket soc, String ansv) {
        super();
        this.soc = soc;
        this.ansv = ansv;
        t = new Thread(this);
        t.start();
    }
```

Создаем и запускаем новый поток

Получаем входной и выходной поток

```
    public void run() {
        try (InputStream is = soc.getInputStream(); OutputStream os = soc.getOutputStream();
            PrintWriter pw = new PrintWriter(os)) {
            byte[] rec1 = new byte[is.available()];

            is.read(rec1);
```

Считываем запрос

```
            String response = "HTTP/1.1 200 OK\r\n" + "Server: My_Server\r\n" + "Content-Type: text/html\r\n" + "Content-Length: " + "\r\n" + "Connection: close\r\n\r\n";
            pw.print(response + ansv);
            pw.flush();
```

```
        } catch (IOException e) {
            System.out.println(e.toString());
        }
```

Отправляем ответ

```
}
```

## Класс Student

```
package com.gmail.tsa;
```

```
public class Student {
```

```
    private String name;
```

```
    private String lastname;
```

```
    private int course;
```

```
    public Student(String name, String lastname, int course) {  
        super();
```

```
        this.name = name;
```

```
        this.lastname = lastname;
```

```
        this.course = course;
```

```
    }
```

```
    public String getName() {  
        return name;
```

```
    }
```

```
    public void setName(String name) {  
        this.name = name;
```

```
    }
```

```
    public String getLastName() {  
        return lastname;
```

```
    }
```

```
    public void setLastName(String lastname) {  
        this.lastname = lastname;
```

```
    }
```

```
    public int getCourse() {  
        return course;
```

```
    }
```

```
    public void setCourse(int course) {  
        this.course = course;
```

```
    }
```

```
}
```

## Класс Group

```
package com.gmail.tsa;

import java.util.ArrayList;

public class Group {

    private String name;
    private ArrayList<Student> al=new ArrayList<>();

    public Group(String name) {
        super();
        this.name = name;
    }

    public void addStudent(Student a){
        al.add(a);
    }

    public Student[] getGroup(){
        Student[] starr=new Student[al.size()];
        for(int i=0;i<al.size();i++){
            starr[i]=al.get(i);
        }
        return starr;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

```
package com.gmail.tsa;
```

```
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        String answer = "";  
        Group gr = new Group("PN121");  
        gr.addStudent(new Student("Ivanov", "Ivan", 3));  
        gr.addStudent(new Student("Petrov", "Petr", 4));  
        gr.addStudent(new Student("Sidorov", "Sidr", 5));  
        gr.addStudent(new Student("Garbuzov", "Grbuz", 1));
```

Создаем и заполняем группу

```
        answer += "<html><head><title>Student</title> <meta charset='utf-8'></head><body><p>Список студентов группы: "  
        + gr.getName() + "</p><br>";  
        answer += "<table border='2' cellpadding='5' ><tr><th>Фамилия</th><th>Имя</th><th>Курс</th></tr>";  
        Student[] starr = gr.getGroup();  
        for (int i = 0; i < starr.length; i++) {  
            answer += "<tr>";  
            answer += "<td>" + starr[i].getName() + "</td>";  
            answer += "<td>" + starr[i].getSurname() + "</td>";  
            answer += "<td>" + starr[i].getCourse() + "</td>";  
            answer += "</tr>";  
        }  
        answer += "</table></body></html>";
```

Генерируем html файл (строку) с данными группы

```
    try (ServerSocket soc = new ServerSocket(8080)) {  
        for (;;) {  
            Socket clisoc = soc.accept();  
            Client cli = new Client(clisoc, answer);  
        }  
    } catch (IOException e) {  
        System.out.println("Error to server Socket Open!!!");  
    }  
}
```

Создаем серверный сокет и скидываем  
подключившихся пользователей в новый поток.

Переходим по адресу сервера и получаем список группы

Student - Mozilla Firefox

Битва. Герои Войн... x Входящие - tsimbal... x Student x

127.0.0.1:8080 Поиск

Список студентов группы: PN121

Фамилия	Имя	Курс
Ivanov	Ivan	3
Petrov	Petr	4
Sidorov	Sidr	5
Garbuzov	Grbuz	1

## Краткие итоги лекции

В Java для работы с сетевыми соединениями используется пакет `java.net.*`

Работа в сетевыми соединениями представлена классами

- **InetAddress** — работа с IP адресами ресурса
- **URL** — установка соединения с ресурсом
- **URLConnection** — получение потока для доступа к удаленному ресурсу
- **HttpURLConnection** — специализированная версия `URLConnection`
- **Socket** и **ServerSocket** — для установки двунаправленного байтового потока для сетевого соединения

## Дополнительная литература по теме данного урока.

- Герберт Шилдт Java 8. Полное руководство 9-е издание, стр 811 — 827
- Кей Хорстман, Гари Корнелл Библиотека профессионала Java Том 2.Расширенные средства . стр. 191-231.



## Домашнее задание

1. Проверить доступность сайтов указанных в отдельном файле.
2. Написать сервер, который будет отправлять пользователю информацию о системе и номер запроса.
3. Напишите программу, которая выведет в файл все ссылки, которые содержатся в html документе, который будет прислан в результате запроса на произвольный URL.
4. \* Реализуйте веб интерфейс к классу Group. Т.е. на экране браузера должен отобразиться список методов, которые можно сделать с группой. При щелчке на ссылку соответствующего метода он должен выполняться. Результат работы, опять же, в виде html страницы должен возвращаться пользователю.

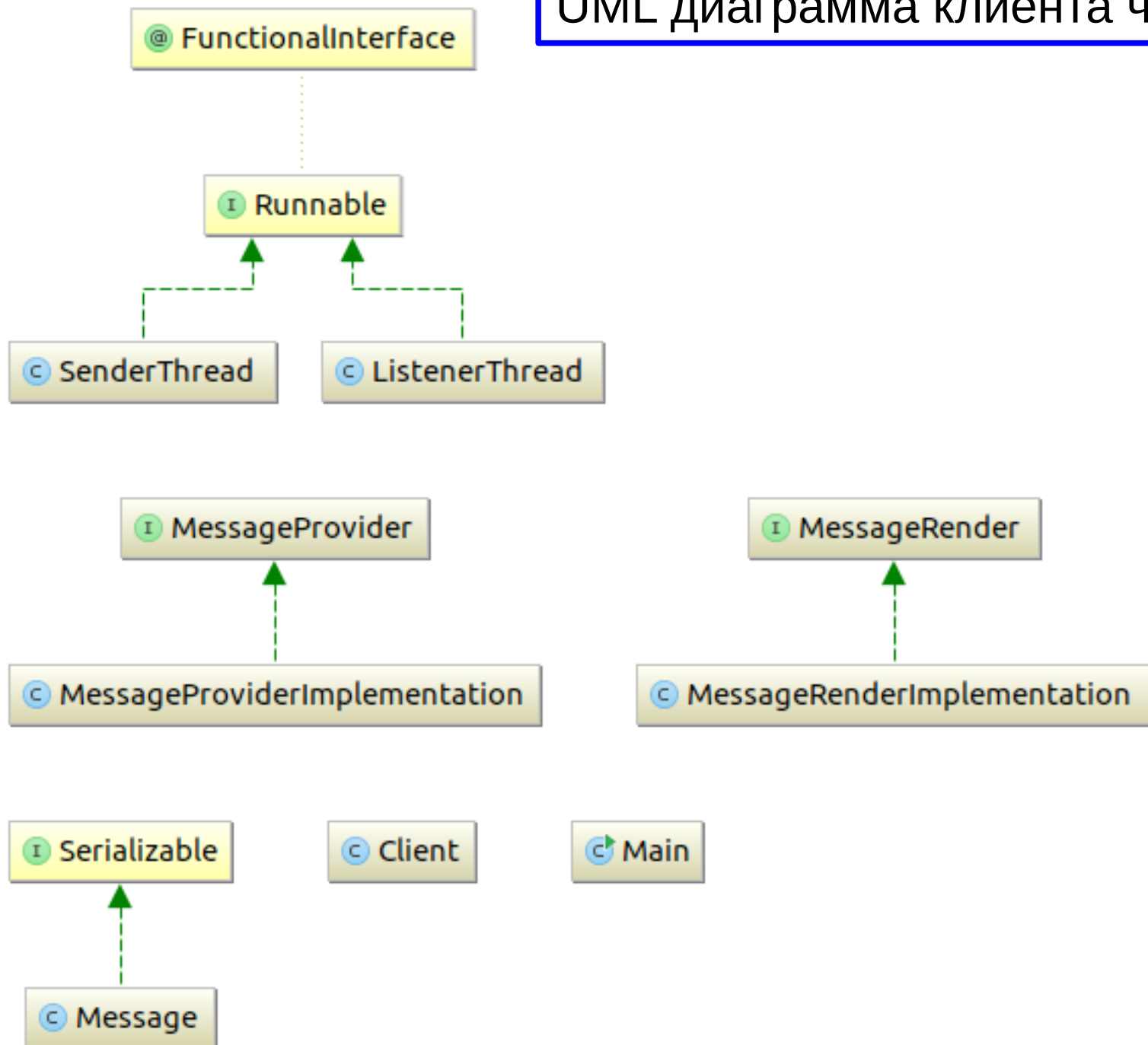
# Java OOP

## (Проект чата на сокетах)

Рассмотрим построение чата на сокетах. Основные приемы и методика..

**Составил: Цымбалюк А.Н.**

## UML диаграмма клиента чата



## Интерфейсы описанные в клиенте чата

Получение и отправление сообщения

```
package com.gmail.tsa;  
  
import java.io.IOException;  
  
public interface MessageProvider {  
    public void sendMessage(Message message) throws IOException;  
    public Message readMessage() throws IOException;  
}
```

Вывод сообщения сообщения

```
package com.gmail.tsa;  
  
public interface MessageRender {  
    public void renderMessage(Message message);  
}
```

## Класс реализующий интерфейс вывода сообщения

```
package com.gmail.tsa;

import java.text.SimpleDateFormat;

public class MessageRenderImplementation implements MessageRender {

    @Override
    public void renderMessage(Message message) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd MMMM yyyy - hh:mm");
        String text = message.getSender() + " " +
            sdf.format(message.getDepartementTime()) + " > " + message.getText();
        System.out.println(text);
    }
}
```

## Класс реализующий интерфейс приема и отправления сообщения. Часть 1

```
package com.gmail.tsa;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class MessageProviderImplementation implements MessageProvider {

    private Socket socet;
    private ObjectInputStream ois = null;
    private ObjectOutputStream oos = null;

    public MessageProviderImplementation(Socket socet) {
        super();
        this.socet = socet;
    }

    public MessageProviderImplementation() {
        super();
    }

    public Socket getSocet() {
        return socet;
    }

    public void setSocet(Socket socet) throws IOException {
        this.socet = socet;
        oos = new ObjectOutputStream(socet.getOutputStream());
        ois = new ObjectInputStream(socet.getInputStream());
    }
}
```

```

@Override
public void sendMessage(Message message) throws IOException {
    try {
        oos.writeObject(message);
    } catch (IOException e) {
        closeStream();
        throw e;
    }
}

```

```

@Override
public Message readMessage() throws IOException {
    try {
        return (Message) ois.readObject();
    } catch (ClassNotFoundException e) {
        return null;
    } catch (IOException e) {
        closeStream();
        throw e;
    }
}

```

```

private final void closeStream() {
    try {
        oos.close();
        ois.close();
        socet.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Класс реализующий интерфейс приема и отправления сообщения. Часть 2

## Класс Message. Часть 1

```
package com.gmail.tsa;

import java.io.Serializable;
import java.util.Date;

public class Message implements Serializable {

    private static final long serialVersionUID = 1L;
    private String text;
    private Date departmentTime;
    private String sender;

    public Message(String text, Date departmentTime, String sender) {
        super();
        this.text = text;
        this.departmentTime = departmentTime;
        this.sender = sender;
    }

    public Message() {
        super();
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```



## Класс Message. Часть 2

```
public Date getDepartmentTime() {  
    return departmentTime;  
}  
  
public void setDepartmentTime(Date departmentTime) {  
    this.departmentTime = departmentTime;  
}  
  
public String getSender() {  
    return sender;  
}  
  
public void setSender(String sender) {  
    this.sender = sender;  
}  
  
@Override  
public String toString() {  
    return "Message [text=" + text + ", departmentTime=" + departmentTime + ",  
        sender=" + sender + "];"  
}  
  
}
```

## Класс-поток для получения сообщений

```
package com.gmail.tsa;

import java.io.IOException;

public class ListenerThread implements Runnable {
    private MessageProvider messageProvider;
    private MessageRender messageRender;

    public ListenerThread(MessageProvider messageProvider, MessageRender
messageRender) {
        super();
        this.messageProvider = messageProvider;
        this.messageRender = messageRender;
    }

    @Override
    public void run() {
        Thread th = Thread.currentThread();
        try {
            for (; !th.isInterrupted();) {
                Message message = messageProvider.readMessage();
                messageRender.renderMessage(message);
            }
        } catch (IOException e) {
            System.out.println(e);
        }
        System.out.println("ListenerThread shutdown");
    }
}
```

## Класс-поток для отправки сообщения. Часть 1

```
package com.gmail.tsa;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Date;

public class SenderThread implements Runnable {
    private MessageProvider messageProvider;
    private String sender;

    public SenderThread(MessageProvider messageProvider, String sender) {
        super();
        this.messageProvider = messageProvider;
        this.sender = sender;
    }

    public SenderThread() {
        super();
    }

    public void setMessageProvider(MessageProvider messageProvider) {
        this.messageProvider = messageProvider;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }
}
```

## Класс-поток для отправки сообщения. Часть 2

```
@Override
public void run() {
    try (BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in))) {
        Thread thS = Thread.currentThread();
        for (; !thS.isInterrupted();) {
            String text = br.readLine();
            Message message = new Message(text, new Date(), sender);
            messageProvider.sendMessage(message);
        }
    } catch (IOException e) {
        System.out.println(e);
    }
    System.out.println("SenderThread shutdown");
}
```

## Класс-Client. Часть 1

```
package com.gmail.tsa;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {
    private String nickName;
    private Socket socet;
    private String serverIP;
    private BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    private MessageProvider messageProvider = null;
    private MessageRender messageRender = null;
    private Thread listenerThread;
    private Thread senderThread;

    public Client() {
        super();
    }

    private boolean init() {
        for (boolean correct = false; correct != true;) {
            try {
                System.out.println("Please input Server IP");
                this.serverIP = br.readLine();
                this.socet = new Socket(serverIP, 20000);
                System.out.println("Input your nickName");
                this.nickName = br.readLine();
                correct = true;
            } catch (UnknownHostException e) {
                System.out.println("Unable to connect. Try another IP");
            } catch (IOException e) {
                e.printStackTrace();
                return false;
            }
        }
    }
}
```

## Класс-Client. Часть 2

```
System.out.println("Initialization start");
MessageProviderImplementation messageProviderImplementation = new
MessageProviderImplementation();
MessageRenderImplementation messageRenderImplementation = new
MessageRenderImplementation();
System.out.println("Set Provider and Render Implementation");
try {
    messageProviderImplementation.setSocet(this.socet);
    this.messageProvider = messageProviderImplementation;
    this.messageRender = messageRenderImplementation;
} catch (IOException e) {
    System.out.println(e);
    return false;
}
this.listenerThread = new Thread(new ListenerThread(this.messageProvider,
this.messageRender));
this.senderThread = new Thread(new SenderThread(messageProvider, nickName));
System.out.println("Initialization end");
return true;
}
public void start() {
    boolean startInit = this.init();
    if (startInit == false) {
        System.out.println("Initialization failed ....");
        return;
    }
    System.out.println("Start chat. Type text and press Enter");
    listenerThread.start();
    senderThread.start();
    for (; listenerThread.isAlive() && senderThread.isAlive();) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

## Класс-Client. Часть 3

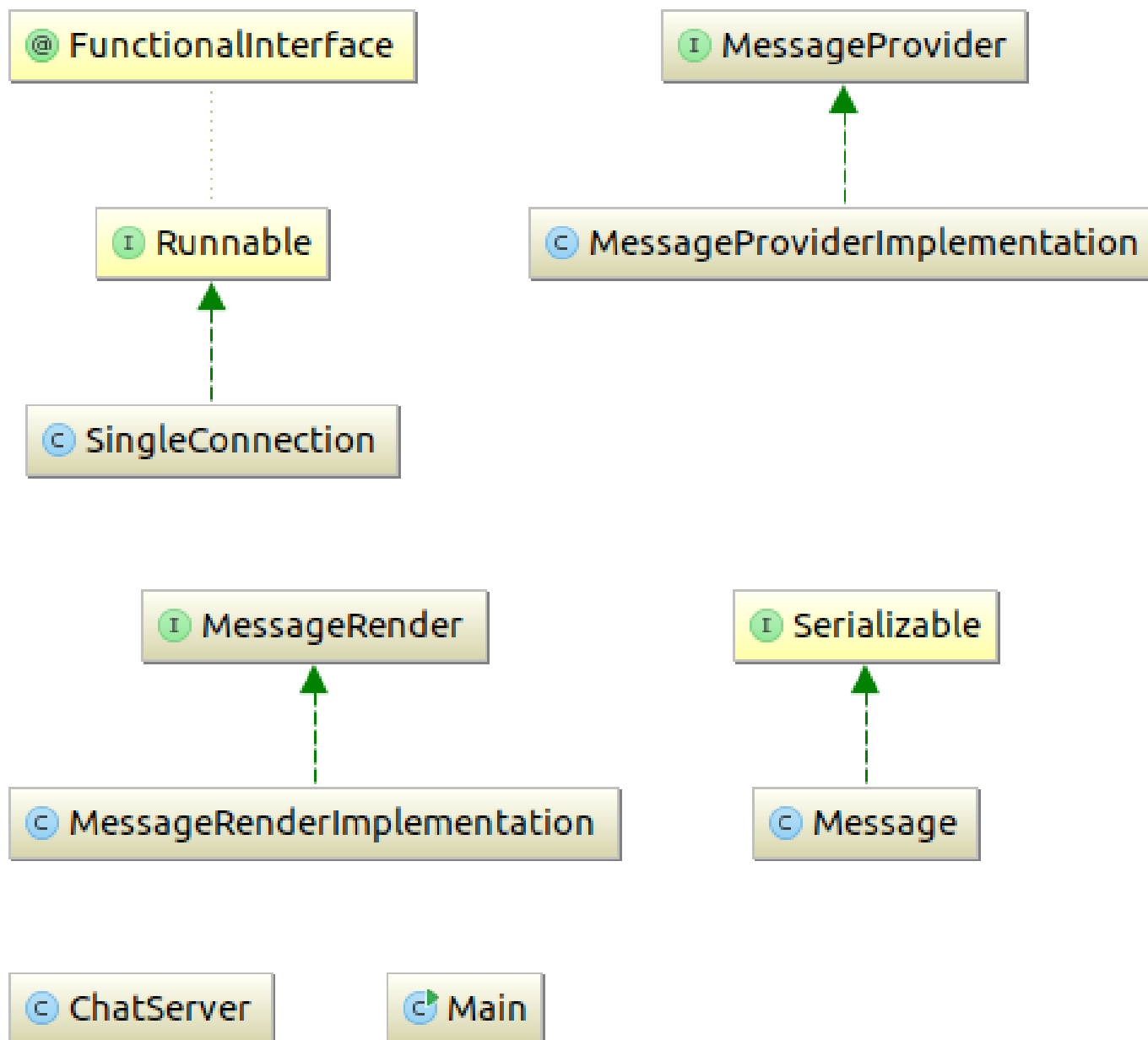
```
    if (listenerThread.isAlive()) {  
        listenerThread.interrupt();  
    }  
    if (senderThread.isAlive()) {  
        senderThread.interrupt();  
    }  
    try {  
        this.socet.close();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
}
```

Main класс проекта.

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Client client = new Client();  
        client.start();  
    }  
}
```



## UML диаграмма сервера чата



## Некоторые общие части с клиентом чата.

Классы сообщения, интерфейсы отправки и вывода не отличаются от таковых для клиента чата.

```
package com.gmail.tsa;
```

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

Класс описывающий единственное соединение с пользователем. Часть 1.

```
public class SingleConnection implements Runnable {  
    private MessageProvider messageProvider;  
    private ArrayList<SingleConnection> connectionList;  
    private List<Message> messageList;  
    private int connectionId;  
  
    public SingleConnection(MessageProvider messageProvider,  
        ArrayList<SingleConnection> connectionList,  
        List<Message> messageList, int connectionId) {  
        super();  
        this.messageProvider = messageProvider;  
        this.connectionList = connectionList;  
        this.connectionId = connectionId;  
        this.messageList = messageList;  
    }  
  
    public MessageProvider getMessageProvider() {  
        return messageProvider;  
    }  
  
    private final void addToConnectionList() {  
        this.connectionList.add(this);  
    }  
  
    private final void deleteFromConnectionList() {  
        this.connectionList.remove(this);  
    }  
}
```

Класс описывающий единственное соединение с пользователем. Часть 2.

```
@Override
public void run() {
    addToConnectionList();
    Thread th = Thread.currentThread();
    try {
        for (Message messageTemp : messageList) {
            this.messageProvider.sendMessage(messageTemp);
        }
        for (; !th.isInterrupted();) {
            Message message = this.messageProvider.readMessage();
            if (message == null) {
                throw new IOException();
            }
            if (message != null) {
                messageList.add(message);
            }
            for (SingleConnection singleConnection : connectionList) {
                singleConnection.getMessageProvider().sendMessage(message);
            }
        }
    } catch (IOException e) {
        System.out.println(e);
        deleteFromConnectionList();
    }
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + connectionId;
    return result;
}
```

Класс описывающий единственное соединение с пользователем. Часть 3.

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    SingleConnection other = (SingleConnection) obj;
    if (connectionId != other.connectionId)
        return false;
    return true;
}
```

## Класс Сервер чата. Часть 1.

```
package com.gmail.tsa;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ChatServer {
    private ServerSocket serverSocket;
    private ArrayList<SingleConnection> connectionList = new ArrayList<>();
    private List<Message> messageList = Collections.synchronizedList(new
        ArrayList<>());

    public ChatServer() {
        super();
    }

    private boolean init() {
        try {
            this.serverSocket = new ServerSocket(20000);
        } catch (IOException e) {
            return false;
        }
        return true;
    }
}
```

## Класс Сервер чата. Часть 2.

```
public void serverStart() {
    boolean startInit = this.init();
    if (startInit == false) {
        System.out.println("Server initialization failed ....");
        return;
    }
    System.out.println("Server start");
    int i = 0;
    for (;;) {
        try {
            Socket socket = this.serverSocket.accept();
            MessageProviderImplementation mpi = new
            MessageProviderImplementation();
            mpi.setSocet(socket);
            SingleConnection singleConnection = new SingleConnection(mpi,
            connectionList, messageList, i++);
            Thread threadSingleConnection = new Thread(singleConnection);
            threadSingleConnection.setDaemon(true);
            threadSingleConnection.start();

        } catch (IOException e) {
            System.out.println("Server stop");
            return;
        }
    }
}
```

## Main класс проекта Сервер чата.

```
package com.gmail.tsa;  
  
public class Main {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        ChatServer chatServer = new ChatServer();  
        chatServer.serverStart();  
    }  
}
```



# Java OOP - Final

Нужно бежать со всех ног, чтобы только оставаться на месте, а чтобы куда-то попасть, надо бежать как минимум вдвое быстрее!

Льюис Кэрролл

**Составил Цымбалюк А.Н.**

# Спасибо за ваше внимание.

Надеюсь, данный курс помог вам в изучении программирования на языке Java. Курс практически полностью покрывает требования Java Core, что является хорошей базой для начала его промышленного применения. В дальнейшем вы можете или заняться web-разработкой Java EE, или разработкой под Android. В любом случае проработка материала данного курса просто необходима.

Если курс вам понравился, можете оставить отзыв или наоборот, если есть замечания, выскажите их нам. Мы всегда открыты для диалога.

Желаю вам удачи в ваших начинаниях.

С уважением Цымбалюк А.Н.