

# Философия Java 2015

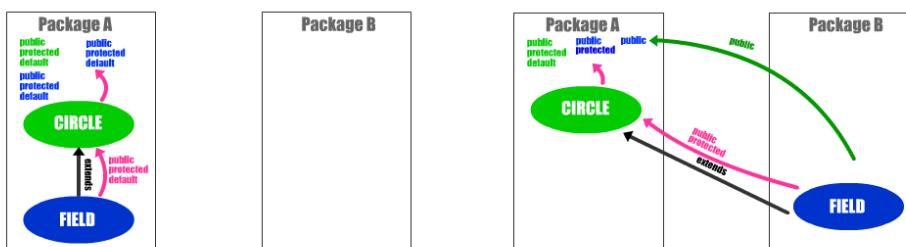
## Объекты

- Тип объекта
  - каждый экземпляр принадлежит какому то классу
- Данные
  - каждый объект содержит в себе другие объекты или примитивы
- Методы
  - каждый объект обладает методами обмена данными
- Интерфейс
  - данные и методы составляют интерфейс объекта

## Инкапсуляция

[ch01/df01/Main](#)

- Ограничение доступа к данным через собственные методы
  - повышение надежности кода
  - упрощение сопровождения кода
- Четыре варианта доступа
  - private                   внутри класса
  - default                 только в рамках пакета
  - protected                только в рамках пакета, но по каналу subclass отовсюду
  - public                   отовсюду



## Композиция или Агрегация

- Внедрение объектов одного класса внутрь объекта другого класса называется композицией
  - называют связь типа «has-a»
  - объекты внутри сразу объявляются private
  - доступ к ним через getter & setter

## Наследование

- Наследование свойств и методов класса родителя классом потомком
- При наследовании свойства и методы родителя расширяются свойствами и методами потомка
  - первый способ это расширение свойств и методов предка свойствами и методами потомка
  - второй способ это переопределение методов предка в потомке
- Свойства наследования
  - возможно только от одного родителя множеству потомков
  - возможно переопределение методов родителя в методах потомков
  - возможно размещать экземпляры потомка в теле ссылки предка это полиморфизм

## Полиморфизм

- Полиморфизм это размещение экземпляров потомков в теле предка
  - ограничение в том, что данному объекту не будут доступны новые методы и свойства потомка
  - зато будут доступны переопределенные методы предка

## Раннее связывание

- При компиляции кода программа сразу знает адрес кода метода который будет выполняться
  - при полиморфизме это не работает, т.к. идет подстановка методов потомка в экземпляр предка
  - применяется позднее связывание

## Позднее связывание

[ch1/ex1/Main](#)

- При полиморфизме адрес метода определяется в момент исполнения
  - при размещении потомка в теле предка, хотя объект предок, выполняться будут методы потомка
  - это позволяет создать массив объектов одного типа, но с различным поведением

## Контейнеры

- Контейнеры это массивы, которые могут хранить объекты разного типа
  - Контейнеры могут иметь разные реализации, массив или список
  - ArrayList массив быстрый произвольный доступ, медленный добавление в середину
  - LinkedList список медленный произвольный доступ, быстрое добавление в середину

## Нисходящее преобразование

- нисходящее преобразование это переход от общего типа объекта к частному, от предка к потомку
- это небезопасное преобразование, неизвестно какой объект будет в конечном итоге
- требует дополнительные затраты на запоминание типа хранимых потомков

## Параметризация

[ch1/ex2/Main](#)

- Параметризация это использование Типа как Параметра при создании экземпляра класса
  - параметризация позволяет ввести ограничение используемых типов объектов
  - параметризованный класс, который позволяет задать конкретный тип объекта хранения

## Generics

- Generics это обобщенный параметризованный тип данных
  - класс с типом Generics имеет механизм уточнения типа данных до конкретного типа
  - при параметризации класса уточнение типа происходит при создании экземпляра
  - задается в угловых скобках
  - при нарушении параметризации ошибка возникает на этапе компиляции

## Иерархия объектов в JAVA Object

- Корневой класс Object
  - это значит что ВСЕ объекты в Java происходят от одного класса Object
  - у всех классов в Java ОДИН предок, это класс Object
  - это упрощает работу GC, так как любой объект в итоге это объект Object, GC просто убирает Object
- **ВНИМАНИЕ.** Корневая структура классов и автоматический GC определяют упрощают программирование

## Поточное программирование

- Использование совместных ресурсов составляет Основную проблему при работе в несколько потоков
  - Java поддерживает механизмы работы с ресурсами на уровне библиотек ???

## **Исключения и Обработка**

- Обработка исключений встроена прямо в язык Java
- Исключение это Объект «Исключение такого то типа» который создается в месте где произошла ошибка
  - после создания объект Исключение перехватывается Обработчиком «Исключений данного типа»
- Обработка исключений это параллельный путь выполнения программы
  - код обработки ошибок НЕ смешивается с кодом программы
  - ВОЗМОЖНО нормальное выполнение программы после Обработки Исключения
  - Исключение НЕВОЗМОЖНО проигнорировать или пропустить
  - Исключение ОБЯЗАТЕЛЬНО будет где то обработано

## **Время жизни объектов**

- Объекты в Java создаются динамически на этапе работы программы в области Heap
- Объект в Java живет до тех пор, не будет уничтожен автоматически Garbage Collector(GC) сборщик
  - GC сам определяет когда объект перестал использоваться
  - программист не занимается уничтожением объектов, это делает GC автоматически
- Время жизни объекта определяется временем работы программы
  - выход за пределы видимости не уничтожает объект
  - объект уничтожается автоматически GC
- Время жизни примитивных типов ограничено областью действия
  - выход за пределы видимости уничтожает объект

## Глава 2

### Объекты

- Объект состоит из двух частей ссылки и тела объекта
  - Ссылка на объект это тот идентификатор с именем в программе
  - Тело объекта это область памяти в Heap где располагается объект
- Особенности работы ссылок в Java
  - в Java нет механизма указателей, все ссылки на объекты являются указателями но и все
  - ссылка на объект при создании равна null
  - ссылка на объект это ссылка, которое присвоен адрес конкретного объекта
  - объект может быть создан без ссылки
  - объект без ссылки не существует долго, автоматически уничтожается GC.

### Хранение объектов Java

- Стек в Stack хранятся примитивные типы и ссылки на объекты
- Heap в Heap хранятся все объекты

### Примитивные типы

- Примитивные типы
  - все примитивные типы в Java знаковые
  - классы обертки могут создать в Heap объекты
- Int
  - Boolean Boolean() false
  - Char Character() '\u0000'
  - Byte Byte() 0x00
  - Short Short() 0 (16bit)0x0000
  - Int Integer() 0 (32bit)
  - Long Long() 0L (64bit)
- Float класс обертка Float() 0.0F
- Double Double() 0.0 <D>
- Литералы
  - L тип Long обязательно
  - F тип Float обязательно
  - D тип Double

### Float MIN\_VALUE и MIN\_NORMAL

- представление float, double в обычной форме 0.xxxEnn
  - мантисса в интервале от  $0 \leq m < 1$
  - такое представление включает ноль в диапазон
  - такое представление имеет много вариантов представления числа
  - например 0.1 можно представить как 0.01E1, 0.001E2 и т.д.
- представление float, double в нормализованной форме 1.xxxEnn
  - мантисса в интервале  $1 \leq m < 10$
  - такое представление не включает 0, для него выделен отдельный бит
  - представление однозначное для любого числа
  - диапазон чисел меньше

- **ВНИМАНИЕ.** Примитивные типы инициализируются ТОЛЬКО когда являются свойствами класса
- В методах локальные примитивные типы **НЕ ИНИЦИАЛИЗИРУЮТСЯ**

## Объекты Числа повышенной точности

- BigInteger класс целых числе с бесконечным числом цифр в составе целого числа
- BigDecimal класс вещественного числа с бесконечным числом цифр после запятой

## Автоматическая упаковка примитивных типов

- автоматическая упаковка это неявная инициализация объекта класса обертки значением примитива
  - происходит когда объекту приравнивают значение примитивного типа
  - когда к объекту применяют унарные операторы

```
Character chObject = 'U'; // автоупаковка
Integer intObject = 15; // автоупаковка
```

```
ArrayList <Integer> arrayList = new ArrayList<>();
arrayList.add(2); // упаковка примитива в массив объектов
arrayList.add(5);
for (Integer i: arrayList) {
    System.out.println("arrayList %2%:" + i%2); // унарный оператор работает с объектом
    Integer
}
```

- автоматическая распаковка это неявная операция извлечения значения примитивного типа из объекта
  - происходит когда примитивному типу назначают объект
  - когда объект передается как параметр вместо примитивного типа в метод

```
static int abs(int a) {
    return Math.abs(a);
}
public static void main(String[] args) {
    Character chObject = 'U'; // автоупаковка
    Integer intObject1 = new Integer(25); // создание объекта
// распаковка
    int a = arrayList.get(0); // распаковка через обращение к массиву
    int b = arrayList.get(1);
    char c = chObject; // значение от объекта к примитиву
    int d = abs(intObject1); // распаковка через функцию которая ожидает примитив
    System.out.println(a+" "+b+" "+c+" "+d);
```

## Массивы в Java

- Массивы
  - массивы примитивных типов инициализируются нулевым значением
  - массивы объектов инициализируются ссылками со значением null
  - за пределами массива обращение вызывает исключение которое можно обработать

## Final

- Final это объявление примитивного типа или ссылки на объект константой
  - final для примитивного типа означает константу в прямом смысле слова
  - final для объекта означает что ссылка созданная на этапе компиляции не может быть изменена
  - имена констант записываются заглавными буквами и подчеркиванием CONSTANT\_VALUE
- final пустые константы
  - ВНИМАНИЕ. пустые final поля класса, можно инициализировать в конструкторе класса

## Инициализация Final

lesson\_ch07/ex19/Flower

- Инициализация final возможна в методе и в класса

## Инициализация final в методе

- Инициализация final в методе
  - final примитивного типа или объекта инициализируется при входе в метод
  - final метода инициализируется заново при каждом входе в метод
  - инициализация final может быть прямо в строке объявления или в любом месте метода
  - ВНИМАНИЕ. static не позволено внутри метода

```
void getLeaf() {  
    final int k = rnd.nextInt(100);           // в строке объявления  
    final Leaf leafG = new Leaf("green"); // в строке объявления  
    // leafY = leafG;                      // не работает  
    final Leaf leafN;                     // объявление final без инициализации  
    int i;  
    i = k+1;  
    leafN = new Leaf("noColor");          // инициализация в другом месте  
}
```

## Инициализация final в классе

- final примитивного типа или объекта в классе инициализируется при входе в класс
  - в строке объявления, в блоке {}, в конструкторе
- final static примитивного типа или объекта в классе инициализируются при первом вызове класса
  - в строке объявления, в блоке static {}
  - ВНИМАНИЕ. удобно проверить с помощью метода Random()

```
public class Flower {  
    final int p = 1; // инициализация в строке  
    final static int m = rnd.nextInt(100); // инициализация в строке  
    final static int nStatic = 1; // инициализация в строке  
    final static int mStatic;  
    static{  
        mStatic = 2;      // инициализация в блоке static  
    }  
  
    {  
        q = 2;          // инициализация в обычном блоке  
    }  
    public Flower() {  
        i = 27; // инициализация в конструкторе константой  
        System.out.println("Flower.Flower() "+i);  
    }  
    public Flower(String s, int i, int j) {  
  
        this.s = s;  
        this.i = i; // инициализация в конструкторе по параметрам  
        this.j = j;  
        System.out.println("Flower.Flower(s,i,j) "+i);  
    }  
}
```

## Неизменные аргументы

## lesson\_ch07/ex19/Flower

- final параметры метода это неизменные аргументы внутри метода
  - получают значение на входе в метод и не могут быть изменены внутри метода
  - ```
void getLeaf( final int i) {
    final String sTemp = "gold leaf";
    System.out.println("final sTemp:" + sTemp+ " input i: "+i);
//      i = 12; // не пропускает
}
void getLeaf(final Leaf leaf) {
    final String sTemp = "gold leaf";
    System.out.println("final sTemp:" + sTemp+ " input Leaf: "+leaf);
Leaf leaf1 = new Leaf("nLeaf");
//
leaf = leaf1; // не пропускает
}
```

## Неизменные методы

- Неизменные методы это final методы которые нельзя переопределить при наследовании
  - final методы нужны чтобы зафиксировать реализацию при наследовании
  - final методы в прошлом считалось повышают эффективность, сейчас неактуально
- Модификаторы final и private в чем разница  
lesson\_ch07/ex20/Flower
  - и private и finale не дают возможность менять метод класса, но разница в доступе
  - final фиксирует метод при наследовании, но он видим и его можно использовать
  - private скрывает метод предка у потомка это просто новый метод
  - **ВНИМАНИЕ.** private не решает проблему, метод предка скрыт, метод потомка просто новый метод
  - **ВНИМАНИЕ.** Директива @Override помогает распознать проблему
  - **Пример**

```
public class Plant {
    final void getGrow() { // фиксирован но доступен
        System.out.println("Plant.getGrow."+this);
    }
    private void getOpen() { // фиксирован но недоступен
        System.out.println("Plant.getOpen."+this);
    }
    public Flower(String s) {
        this.s = s;
        System.out.println("Flower.Flower()");
    }
    // @Override
    // void getGrow() { // невозможно так как метод final
    // }
    @Override // в компиляторе показывает что нет переопределения
    void getOpen() { // сам метод нормально компилируется и работает
        System.out.println("Flower.getOpen."+this);
    }
}
```

## Неизменные классы

- Неизменный класс это класс от которого нельзя наследоваться
  - класс не может иметь потомков
  - все методы класса final по умолчанию, нельзя переопределить т.к. класс не может быть предком
  - все поля класса могут быть, а могут не быть final на выбор

## Final Предостережения

- **ВНИМАНИЕ.** реже применять final так как
  - final запрещают модификацию полей, методов, классов которая может потребоваться в будущем
  - final могут зафиксировать неоптимальную реализацию метода которую невозможно исправить

# Синтаксис Java

## Видимость классов

- Видимость классов
  - **import** импорт пакета или библиотеки чтобы обеспечить видимость
  - имя пакета пишется с строчной буквы, запись доменного имени в обратном порядке
  - имя класса пишется с Заглавной буквы

## Статические объекты, методы и свойства

ch1/ex4/Main

- Статический объект
  - когда нужен объект хранящийся в единственном экземпляре
  - когда доступ к объекту нужен из любого экземпляра или из класса
- Статический метод
  - нужен чтобы доступ к методы был из класса без создания экземпляров класса
  - используется в абстрактных классах
- Статический объект необходим
  - когда нужен объект хранящийся в единственном экземпляре

## JavaDoc Документация

- Правила оформления документации JavaDoc
  - **/\*\*** начало комментария всегда две звездочки
  - **\*** в каждой строке по звездочке
  - **\*/** завершение комментария JavaDoc
  - **public** видимость обязательна для включения в JavaDoc
  - **protected** видимость поддерживается JavaDoc
  - **default** поддерживается с уровнем package private **задать в IDEA** package private
  - **private** поддерживается с уровнем private **задать в IDEA** private
  - **комментарий** для класса, метода и свойства размещается в предыдущей строке
- Автономные теги @
  - **@param** ключевое слово для включения реальных параметров метода
  - **@throws** ключевое слово для exception метода
  - **@see** ссылка на другие классы
  - **@link** package.Class#class\_member Label ссылка на другой класс
  - **@docRoot** путь к корневой папке
  - **@inheritDoc** наследует документацию ближайшего к данному классу
  - **@version** задать версию Вы ее определяете **задать в IDEA** version
  - **@since** задать версию кода с которой началось использование данной возможности
  - **@return** описание параметров возвращаемых методом
  - **@author** автор **задать в IDEA** author
- HTML теги <>
  - в комментарии можно вставлять HTML теги и они работают
  - **<ol> </ol>** список пунктов
  - **<li>** пункт списка **<ol> </ol>**
  - **<pre> </pre>** блок форматированного текста в несколько строк
  - **<em> </em>** напечатать курсивом
  - **<br>** перевод строки

- **ВНИМАНИЕ.** Для компиляции русских символов ввести в окне Other arguments команду «-encoding utf8»

- Пример оформления комментариев JavaDoc

```
import java.util.Date;
/** HelloWorld comment for Class HelloWorld
 * * <br> <br>
 * You can add <em> add HTML list </em> into comment:
 * <ol>
 * <li> First item
 * <li> Second item
 * <li> Third item
 * </ol> <br><br>
 * * <pre>
 * System.out.println(new Date()); example of command description
 * This is preformatted block of text
 * You can write as ordinary text and place here
 * </pre>
 * @version 1.223/254
 * @author V1
 * @author Added comment for author company
 * */
public class HelloWorld {
    /** iClass comment I to static variable i */
    public static int iClass;

    /** mClass comment M to variable m
     * @since 1.223/001
     * */
    protected int mClass;

    /** nClass comment N to variable n */
    private static int nclass;

    /** pClass comment P default to variable p */
    int pClass;

    /**
     * Hello World constructor description
     * @param mClass description for mClass
     * @param pClass description for pClass
     */
    public HelloWorld(int mClass, int pClass) {
        this.mClass = mClass;
        this.pClass = pClass;
    }

    /** @param iVoice comment I to method voice()
     * @param kVoice comment K to method voice
     * @return int comment for return value
     * @see HelloDate HelloDate class comment SeeAlso
     * @deprecated method deprecated comment
     */
    public static int voice(int iVoice, int kVoice) {
        System.out.println(iVoice+" "+kVoice);
        return iVoice+kVoice;
    }
    /** @param iShow comment I to method show */
    public void show(int iShow) {
        System.out.println(iShow);
    }

    /** @param args comment to args for method main()
     * @throws Exception comment to exception from method main() */
    public static void main(String[] args) throws Exception {
        int i;
        System.out.println("Привет, сегодня: ");
    }
}
```

}

## Операторы

- Оператор получает один или несколько аргументов и выполняет операцию
  - При делении
  - + сложение
  - - вычитание
  - \* умножение
  - / деление
  - = присваивание
  - +=, -=, \*=, /= комбинированные операторы
  - ++ инкремент
  - -- декремент
- Свойства операторов
  - почти все операторы работают с примитивами
  - операторы работающие с объектами
  - =
  - ==
  - !=
  - операторы работающие со String
  - +
  - +=
  - оператор / не округляет результат, производит усечение
- Присваивание для примитивных типов
  - при присвоении примитивных типов идет передача значения
  - при присвоении между примитивным типом и объектом идет упаковка, распаковка
- Присваивание для объектов
  - при присвоении объектов идет передача ссылки на объект, происходит совмещение имен
- Bit operators
  - & bitwise AND
  - | bitwise OR
  - ^ bitwise XOR
  - ~ bitwise NOT
  - **ВНИМАНИЕ.** Способ печати двоичных чисел смотреть в ch02/ex10/Ex10
    - Integer.toBitString() стандартный способ
    - printf("%08d", Integer.valueOf(Integer.toBitString(int))) не стандартный способ
- Операторы сдвига
  - >> сдвиг вправо с заполнением старших битов знаком pos > 0 , neg > 1
  - << сдвиг влево
  - >>> беззнаковый вправо заполнение старших битов всегда 0
  - **ВНИМАНИЕ.** Оператор >>> не работает с типом Short и Byte

```
short s = -1;
System.out.println(Integer.toBinaryString(s));
s>>>=4;
System.out.println(Integer.toBinaryString(s)); // >>> не работает с типом Short
byte b = -1;
b>>>=4;
System.out.println(Int.toBinaryString(b)); // >>> не работает с типом Byte
```

- Тернарный оператор
  - оператор вида  $(a==b)? x:y;$       реализует запрос if/else в одну строку

## Совмещение имен

- Совмещение имен при присваивании объектов
  - при присвоении объектов идет перезапись ссылки на объект
  - в результате две ссылки указывают на один и тот же объект, а второй при этом потерян
  - обе ссылки будут менять тот же объект

```
public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    t1.level = t2.level; // присвоение примитивных типов
    t1 = t2; // перезапись ссылок, обе ссылки указывают на t2
    t1.level = 21; // изменит значение на 21 и обе ссылки покажут 21
    t2.level = 32; // изменит значение на 32 и обе ссылки покажут 32
```

- Совмещение имен при передача объекта методу
  - когда объект передается в метод, то все присвоения полям в методе производятся с оригиналом
  - **ВНИМАНИЕ.** Внутри нельзя перезаписать значение ссылки оригинала принятую как параметр
  - ну то есть если  $a = b$  на входе в метод и затем  $a = c$  то это не значит что  $b$  изменится

```
public PassObject() {
    xField = new Letter();
}
void f(Letter y) { // передали объект по ссылке
    y.c = 'z'; // объект изменяется вот тот оригинал Letter x
    y = xField; // это поле не работает, т.к. y=b > y=c не означает что b=c
}
void passMethod() {
    Letter x = new Letter();
    x.c = 'a';
    xField.c = 'b';
    System.out.println("x.c = a > x.c: "+x.c);
    f(x);
    System.out.println("f(x) > x.c: "+x.c); // перезаписали поле с объекта x
```

## Random генератор

- Диапазон значений
 

|                    |            |                                                       |               |
|--------------------|------------|-------------------------------------------------------|---------------|
| ○ nextInt(10)      | -9..0..9   | без аргументов                                        | -Int...+Int   |
| ○ nextLong(10)     | -9..0..9   | без аргументов                                        | -Long...+Long |
| ○ nextFloat()      | 0...0.9(9) |                                                       |               |
| ○ nextDouble()     | 0...0.9(9) |                                                       |               |
| ○ nextBits(1..32)  |            | генерит заданное количество бит и помещает их в (int) |               |
| ○ setSeed(10L)     |            | задает состояние генератора, seed типа long           |               |
| ○ nextBytes(array) |            | заполняет массив байт                                 |               |
| ○                  |            |                                                       |               |

## Increment Decrement

- определение инкремента
  - `i++` провести операцию затем увеличить на 1      инкремент
  - `++i` увеличить на 1, затем провести операцию      инкремент
- определение декремента
  - `i--` тоже самое уменьшить на 1      декремент
  - `--i` тоже самое уменьшить на 1      декремент
- Поведение инкремента в смешанных операции
  - первое надо разделять понятие текущей суммы и значение переменной в операциях `++i`, `i++`
  - `+=` для суммирования используется начальное значение до входа в уравнение
  - `++i` сначала идет инкремент затем участие в операции
  - `i++` сначала переменная участвует в операции, затем наращивается но уже для другой
  - Пример с пояснениями

```
i = 5;
i += ++i + i++ + ++i; // движение слева направо
// 1. ++i => 5+1 = i=6 s=i => i=6 s=6
// 2. (s) + i++ => s+i = 6 + 6 = s=12 i++=7
// 3. (s) + ++i => i = 7+1=8 => s + i => 12+8 = 20 => s=20 i = 8
// 4. i+=s i=5(начальное значение) s=20 => i = i+s => i = 5+20 = 25
System.out.println(i);
i=5;
i *= ++i * i++ * ++i; // движение слева направо
// 1. ++i => 5+1 = i=6 s=i => i=6 s=6
// 2. (s) * i++ => s*i = 6 * 6 = s=36 i++=7
// 3. (s) * ++i => i = 7+1=8 => s * i => 36*8 = 288 => s=96 i = 8
// 4. i*=s i=5(начальное значение) s=288 => i = i*s => i = 5*288 = 1440
System.out.println(i);
```

## Сравнение примитивов и объектов

- сравнение примитивных типов идет по значению
- сравнение объектов идет по ссылкам на объекты плюс дополнительные условия
  - `==` сравнение объектов по ссылке по сути это реализация `Object.equals()`
  - **ВНИМАНИЕ.** сравнение по ссылке и не имеет отношения к значению объекта
  - `new Integer(5) != new Integer(5)` так как несмотря на значение ссылки разные
  - `equals ()` сравнение объектов по конкретным для данного класса полям или условиям  
`(new Integer(5)).equals(newInteger(5))` сравнивает значения
  - **ВНИМАНИЕ.** При переопределении `equals` обязательно запустить механизм `override()`

## Приведение типа

- Приведение типа это изменение типа данных например `long l = (long)25;`
- Приведение типа бывает сужающим и расширяющим
  - сужающее приведение `short c = (short) intValue` является опасным, требует явного приведения типа
  - расширяющее приведение `int c = shortVal` является безопасным и не требует явного приведения
- Приведение типа с усечением
  - при переходе от `float,double` к целочисленным `short/int/long` происходит усечение дробной части

```
int a = (short) (int) (new Integer(2)); // явное приведение типа
short c = a; // не работает сужающее неявное приведение типа
c = (short)a; // работает сужающее явное приведение типа
int b = c; // работает расширяющее неявное приведение типа
```

## Логические операции

- Логические операторы
  - операторы > < >= <= != !
  - результат всегда true и false, можно объединять true==true или !false == true
  - **ВНИМАНИЕ.** Не путать оператор ! “НЕ” с оператором ~ “инверсия”
  - **ВНИМАНИЕ.**  $\sim 0 = -1$  инверсия в обе стороны работает по принципу дополнения :  $\sim 0 = 0-1 = -1$
- Операторы объединения
  - операторы && || объединяют операторы сравнения
  - && логическое И
  - || логическое ИЛИ
  - например (a==b) && (b==c) , (a!=b) || (a==c)
- Ускоренное сравнение
  - если в цепочке стоит несколько операций сравнения, сравнение прекращается если результат ясен
  - Пример
    - ```
boolean cmp = circuit.test1(0)&& circuit.test2(2)&&circuit.test3(2);
System.out.println("calculation result: "+cmp);
```
    - // остановится на шаге 2 т.к.  $2 < 2 = \text{false}$  и  $\text{cmp} = \text{false}$
    - // независимо от значения шага 3 все сравнения по && И
    - // результаты вывода на печать
    - // Speedup calculation
    - //
    - // test1(0)> result: true
    - // test2(2)> result: false
    - // calculation result: false

## Циклы While и Labels

lesson\_ch04/ch/ex8/Ex8

- Label ставят только перед началом цикла прямо перед словом for, while, do
  - при переходе на метку цикл просто продолжает свою работу
  - метки работают как continue но позволяют «выпрыгивать» из внутренних циклов
  - continue продолжает текущий цикл
  - continue Label продолжает цикл текущий или верхнего уровня с меткой Label
  - break выходите из текущего цикла
  - continue Label выходит из текущего цикла или цикла верхнего уровня с меткой Label
  -

```
label1:
while (i++ < 150) {
    if (i % 9 == 0) {
        continue label1;           // работает как continue;
    }
    label2:
    for (int j = 0; j < 100; j++) {
        if (i % 15 == 0) {
            continue label1;       // выпрыгивает во внешний цикл
        }
        if (j % 9 == 0) {
            continue label2;       // работает как обычный continue
        }
        if (i == 132) {
            break label1;          // выход из внешнего while
    }
}
```

## Конструкторы

- Самые проблемные участки кода это инициализация и завершение
  - неверная инициализация переменных или компонентов библиотеки
  - неверное освобождение ресурсов при завершении
- Синтаксис
  - конструктор называется также как и класс
  - имя конструктора начинается с большой буквы
  - у конструктора нет возвращаемого значения
  - класс можно создать без конструктора, в этом случае конструктор неявно создается автоматом
  - если есть хотя бы один конструктор, неявный конструктор не создается
  - если единственный конструктор с параметрами, экземпляр можно создать только с параметрами
  - this() вызов конструктора из метода класса, только один конструктор и только в 1й строке метода

```
public Turbo(String s) {
    this.s = s;
    System.out.println("String Constructor");
}

public Turbo(int i) {
    this("String init"); // только 1 конструктор и только в 1й строке
    this.i = i;           // class i и local i
    System.out.println("Int Constructor");
}

public Turbo() {
    this(1); // конструктор Int
    this.m = 1;
    System.out.println("Default Constructor");
}
```

- Оператор this

- this              указатель на экземпляр класса внутри экземпляра класса
- this.method()      вызов метода экземпляра
- this.field        обращение к полю класса, актуально при инициализации
- return (this)     возвращает ссылку из метода на экземпляр который его вызвал
- **ВНИМАНИЕ.**    Это удобно когда надо вызвать цепочку методов в одной строке, пункты не больше  
                        Еще как результат обработки статическим методом, но смысла большого нет

```
Turbo turbo = new Turbo(); // получить экземпляр
Turbo t2=null;           // пустая ссылка даже не null

t2 = turbo.getInstance(5); // получить копию экземпляра класса
t2.getIcrement().getIcrement().getIcrement(); // инкремент вернуть ссылку на экземпляр

public class Peeler {
    static Apple getPeeled(Apple apple) {
        apple.apple--;
        return apple;
    }
}
public class Apple {
    int apple;
}

Apple apple = new Apple();
Apple apple1 = Peeler.getPeeled(apple);
```

## Освобождение ресурсов

- Освобождение ресурсов
  - все объекты созданные оператором new освобождаются Garbage Collector GC уборщиком мусора
  - finalize() предназначался для освобождения ресурсов занятых внешним кодом от C/C++
  - **ВНИМАНИЕ.** Есть ситуации когда GC не помогает надо создать свои методы освобождения памяти
  - **ВНИМАНИЕ.** Не использовать метод finalize() из-за неизвестной вероятности его реального вызова
  -
- Метод finalize() использование для проверки «условия готовности»
  - условие готовности это готовность объекта к освобождению памяти
  - GC вызывает «с вероятностью» метод finalize() для всех объектов, которые потеряли свои ссылки
- Проверки «условия готовности»
  - общий смысл, когда бы не был вызван метод finalize() он покажет отсутствие очистки объекта
  - Class >> boolean check >> Constructor() >> check = true >> Override finalize() >> if(check) sout "Error"
  - создать поле «check»
  - задать в конструкторе его значение «true» что означает объект создан в памяти корректно
  - создать метод который сбрасывает флаг «check» если выполнены все процедуры очистки объекта
  - переопределить finalize() и выдать ошибку если флаг check установлен в true
  - **ВНИМАНИЕ.** При тестировании возможно потребуется задержка на 100ms
  - Надо задержать поток, иначе он закроется раньше вывода результатов finalize()

```
public class Book {  
    boolean checkedOut = false;           // при создании false Book()  
    public Book(boolean checkedOut) { //Book(true) программист задает флаг занятости true  
        this.checkedOut = checkedOut;  
    }  
    void checkIn() {                     // ручная очистка объекта  
        checkedOut = false;             // программист сбросил поле вручную false  
    }  
    @Override  
    protected void finalize() throws Throwable { // переопределение finalize()  
        if (checkedOut) {                // проверяем была ли ручная очистка объекта  
            System.out.println("Ошибка: checkedOut > Book "+this);  
        }  
        super.finalize();              // уходим на стандартный finalize()  
    }  
    public Book() {  
    }  
}  
  
Book book = new Book(true); // создали объект флаг установлен  
System.out.println("\nCheck Ready Condition \n");  
System.out.println(book.checkedOut);  
book.checkIn();           // флаг сбросили "очистили" объект  
new Book(true);          // создали новый объект флаг установлен  
new Book(true).checkIn(); // создали новый объект и тут же флаг сбросили  
System.gc();  
try {                    // задержка чтобы увидеть результаты работы GC  
    Thread.sleep(100);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

## Garbage Collector

- GC работает адаптивно, использует методы «пометить и удалить» и «остановиться и копировать»
  - адаптивная работа означает что при сильной дефрагментации «остановиться и копировать»
  - если мусор накапливается медленно, GC переключается в режим «пометить и удалить»
  - уборка мусора это удаление мусора и упаковка живых объектов в компактно в блоки памяти
- метод «пометить и убить»
  - быстрый метод при небольшом уровне генерации мусора
  - суть, что новый мусор только помечается на удаление, но реально живет в системе до уборки
  - уборка происходит только выборочно внутри блока памяти, где это возможно
  - при малом количестве мусора метод работает быстро
- метод «остановиться и копировать»
  - все процессы останавливаются и уборщик мусора начинает упаковывать живые объекты в блоки
  - метод работает медленно, практически останавливает JVM это оправданно при нехватке памяти
  - когда мусора нет, метод все равно перекачивает память туда сюда
- ускоренные методы GC
  - JIT Just In Time компиляция кода во время загрузки байт кода в код JIT без участия JVM
  - JIT код создается если будет выполняться, в итоге часто вызываемый код выполняется быстрее

## Инициализация с наследованием и полиморфизмом

- Общий порядок загрузки класса
  - класс хранится в отдельном файле и класс загружается в точке первого обращения к классу
  - обращение к классу либо через static поля, методы, либо через вызов конструктора класса
- Порядок инициализации класса с наследованием lesson\_ch07/ex23/Ex23, ex24/Ex24
  - предок        static переменные в        месте объявления
  - предок        static переменные в        static блоке
  - потомок      static переменные в        месте объявления
  - потомок      static переменные в        static блоке
  - предок        переменные в                месте объявления
  - предок        переменные в                блоке
  - предок        конструктор                в первой строке можно разместить super()
  - потомок      переменные в                месте объявления
  - потомок      переменные в                блоке
  - потомок      конструктор                в первой строке можно разместить super()

## Инициализация класса

- Общий порядок инициализации класса
  - static поля класса                        в месте объявления при любом первом обращении к классу
  - static блок класса                        выполняется при любом первом обращении к классу
  - обычные поля класса                    в месте объявления инициализируются ДО конструктора
  - обычный блок класса
  - конструктор класса

## Что надо помнить при инициализации

- Нельзя создавать static объекты в конструкторе класса
  - **ВНИМАНИЕ.** static НЕЛЬЗЯ инициализировать КОНСТРУКТОРЕ создает МУСОР и ПОТЕРЮ ДАННЫХ
  - создание мусора новый экземпляр пересоздает static объект, выбрасывает предыдущий
  - потеря данных      новый экземпляр пересоздает static объект, данные предыдущего теряются

## Инициализация полей

- Инициализация членов класса происходит автоматически
  - члены класса инициализируются нулевым или null автоматически
  - члены класса можно инициализировать прямо в строке объявления или в конструкторе
- Инициализация членов методов должна проводиться явно
  - члены метода обязаны быть инициализированы явно

## Инициализация обычным блоком

- выполняется один раз при создании экземпляра класса
- выполняется до конструктора, гарантирует выполнение до любого перегруженного конструктора
- для поддержки анонимных внутренних классов

```
System.out.println("\nAnonymous Class Check\n");
class Mbr {
    int i;
    {
        i = 15;
    }
    Mbr(int i) {
        this.i = i;
    }
    void show() {
        System.out.println(" "+this.i);
    }
}
Mbr m = new Mbr();
Mbr k = new Mbr(10);
m.show();
k.show();
```

## Инициализация полей static

- Инициализация static всегда происходит автоматически
  - static всегда только члены класса поэтому инициализируются нулевым значением или null
  - static инициализируются один раз при первом обращении к классу, неважно экземпляр или класс
  - static объект можно создавать или инициализировать переменную явно в строке
  - static объект можно создавать или инициализировать переменную в блоке static
  - **ВНИМАНИЕ.** static НЕЛЬЗЯ инициализировать КОНСТРУКТОРЕМ создает МУСОР и ПОТЕРЮ ДАННЫХ
  - создание мусора новый экземпляр пересоздает static объект, выбрасывает предыдущий
  - потеря данных новый экземпляр пересоздает static объект, данные предыдущего теряются

```
public Cups() {
    System.out.println("static cup2 initialized by Cups()");
    System.out.println("Cups");
//    cup2 = new Cup(2); //некорректное размещение выдает мусор и потеря данных на
//    //каждом экземпляре
```

- Порядок инициализации static при обращении к классу, вызов метода или поля
  - сначала static поля класса, затем конструктор класса
- Порядок инициализации static при создании экземпляра класса
  - сначала static поля класса, затем обычные поля класса, затем конструктор класса

## Инициализация static блоком

- static блок выполняется один раз при первом обращении к классу, работает только с static полями
- выполняется до конструктора, гарантирует выполнение до любого перегруженного конструктора

```
static {
    cup1 = new Cup(1);
    cup2 = new Cup(2);}
```

## Инициализация массивов

- Организация массивов
  - массивы могут содержать объекты только одного типа, объекты класса или примитивные типы
  - массив состоит из ссылки и тела массива
  - int[] name это ссылка на массив
  - new int[10] собственно массив
- Синтаксис
  - int [] name = new int[12]
  - int [] name = new int[]{ 1,2,3,4,5 }
  - Integer[] name = new Integer[10];
  - Integer[] name = new Integer[10];
- Полная форма записи
  - порождает объект и можно передавать в метод как новый массив

```
int[] intd = new int[] {1,2,3,4,5,6,7,8};           // массив примитивов
Integer [] intc = new Integer[] {new Integer(1),new Integer(2),new Integer(3)} ;
```

- Сокращенная форма записи
  - **ВНИМАНИЕ.** Объявление без new содержит его но можно использовать только в точке объявления

```
int[] intf = {1,2,3,4,5,6,7,8};           // массив примитивов
Integer [] intg = {new Integer(1),new Integer(2),new Integer(3)} ;
```

- Создание массивов примитивного типа
  - массивы инициализируются нулем, и доступны к использованию сразу
  - int [] name = new int[12] массив содержит элементы с значением 0, можно работать
  -
- Создание массивов объектов классов
  - массив состоит из ссылок null, нужна инициализация элементов , иначе сработает исключение
  - Integer[] name = new Integer[10]; массив содержит элементы null, нужна инициализация

```
int[] ints = new int[10];                  // массив примитивов {0}
int[] intd = new int[] {1,2,3,4,5,6,7,8}; // массив примитивов
Integer [] intObj = new Integer[10] ;       // массив объектов неинициализированный
Integer [] intb = new Integer[] {1,2,3,4,5} ; // массив объектов автоупаковка
Integer [] intc = new Integer[] {new Integer(1),new Integer(2),new Integer(3)} ;
Msg[] msgr = new Msg[10];
for (int i = 0; i < msgr.length; i++) {
    msgr[i] = new Msg("Message Number "+ i); // конструкторы сработали
```

## Массивы списки переменной длины Object

- Автоматическое определение длины массива в аргументе типа Object[]
  - <Object ...> позволяет принять в метод Object[], автоматически распознать тип длину массива
  - <Stringt ...>, <Character...>, <Integer ...> позволяет автоматически распознать тип и длину массива
  - **ВНИМАНИЕ.** Перегрузка может не работать со списком параметров, например если аргумент пуст

- Работа со списком разных аргументов
  -

```
static void print(Object... iArray) {    // список аргументов любой длины любого типа
    for (Object o : iArray) {
        System.out.print( o + " ");
    }
    System.out.println();
}

Msg.print();
Msg.print(1, 2, 3, 4);
Msg.print(1, "str", 2.71, 35.43, 'c' + (short) 4);
Msg.print(new Integer(15), new Float(2.54), new String("SetData ") +
    new Character('\uF0FF'));
```

- Работа с одномерным массивом объектов
  - <Object ...> работает также с одномерными массивами объектов Integer[], Character[]

```
static Character[] toObject(String s) {
    char chs[] = s.toCharArray();
    Character[] charArray = new Character[chs.length];
    for (int i = 0; i < chs.length; i++) {
        charArray[i] = new Character(chs[i]);
    }
    return charArray;
}

static Integer[] toObject(int[] ints) {
    Integer[] intArray = new Integer[ints.length];
    for (int i = 0; i < ints.length; i++) {
        intArray[i] = new Integer(ints[i]);
    }
    return intArray;
}

Msg.print(ArrayUtils.toObject("Character Array ")); // массив Character[]
Msg.print(ArrayUtils.toObject(Range.rangeInt(25))); // массив Integer[]
```

- Работа со списком аргументов необязательным к вводу на базе String
  - можно не вводить вообще строковые аргументы, и это распознается

```
static void out(int i, String... iStr) {    // список аргументов любой длины любого типа
    System.out.print("arguments : "+i+" ");
    for (String s : iStr) {
        System.out.print( s + " ");
    }
    System.out.println();
}

Args.out(2, "-k", "-m");
Args.out(0);
```

- Работа со списком аргументов <Character ...>, <Integer ...>
  - можно вводить массив объектов или массив примитивных типов, либо в перемежку

```

System.out.println("\nCharacter Array Check");
// Character[] chObj = ArrayUtils.toObject("ArrayUtils.toObject ");
// Msg.printChr(chObj);

Msg.printChr(ArrayUtils.toObject("ArrayUtils.toObject "));

System.out.println("\nInteger Array Check");
// int [] ints = Range.rangeInt(25); // получить int[]
// Integer[] intObj = ArrayUtils.toObject(ints); // получить Integer[]
// Msg.printInt( intObj); // вывести func(Integer...)
Msg.printInt(ArrayUtils.toObject(Range.rangeInt(25))); // Integer[]
Msg.printInt(1,2,3,4,5,6); // int [] автоупаковка

static void printObj(Object... iArray) { // список Object[] любой длины любого типа
    for (Object o : iArray) {
        System.out.print( o + " ");
    }
    System.out.println();
}
static void printStr(int i, String... iStr) { // список String[] любой длины
    System.out.print("arguments : "+i+" ");
    for (String s : iStr) {
        System.out.print( s + " ");
    }
}
static void printChr(Character... iChr) { // список Character[] любой длины
    for (Character c : iChr) {
        System.out.print( c.toString() + " ");
    }
}
static void printInt(Integer... iInt) { // список Integer[] любой длины
    for (Integer i : iInt) {
        System.out.print( String.valueOf(i) + " ");
    }
}
static void printInt(Float... iInt) { // список аргументов любой длины любого типа
    for (Float i : iInt) {
        System.out.printf( "%2.1f ",i );
    }
    System.out.println();
}
}

```

## Перечисления

- Перечисления являются классами
  - правила именования перечисления как у класса с большой буквы
  - члены перечисления именуются как константы, заглавными буквами и через подчеркивание
- Методы перечисления
  - `toString()` выводит в String текущий элемент
  - `name()` имя элемента STRING
  - `ordinal()` порядковый номер в перечислении
  - `compareTo()` позицию относительную сравниваемого элемента
  - `equals()` сравнивает равенство элементов
  - `valueOf()` возвращает элемент enum по имени
  - `values()` возвращает массив всех значений перечисления

## Управление доступом

### Разграничение доступа

- Разграничение доступа состоит из двух задач инкапсуляции и ограничения доступа к классам
- Инкапсуляция
  - это ограничение доступа к данным и методам класса из других классов или пакетов
- Ограничение доступа к классам
  - это ограничение области видимости и доступности класса в рамках пакета

### Пакеты

- Класс – это файл с именем класса который содержит внутри одноименный класс
- Пакеты позволяют ограничиваю пространство имен набором выбранных классов
  - по сути это папка где размещаются исходные файлы классов
  - чтобы расширить пространство имен используют директиву `import <package.name.Class>`
  - классы можно подключать напрямую с полным именем, это неудобно, поэтому используют import

### Синтаксис

- package
  - `package <name>` первая строка файла
  - `package ru.site.www.project` путь строится как в домене, сначала домен верхнего уровня
  - `import ru.site.www.project2.*` подключить все классы project2 в данный проект
- class
  - `public class` объявление основного класса в данном файле
  - основной класс должен быть public
  - внутри могут быть другие классы но обязательно без public
  -

### CLASSPATH

- Описание CLASSPATH
  - переменная где могут располагаться файлы проекта [проверено video java 001.camproj](#)
  - пусть проект в C:\prj\src\\*.java , чужие исходники в net/data/\*.java, CLASSPATH=".;..;C:\path\code"
  - проект сработает если чужие исходники будут в C:\prj\net, C:\prj\src\net, C:\path\code\net
- Подключение исходников
  - Распаковать исходники в папку .../code
  - Создать модуль code из исходников в папке .../code
  - File >> New >> Module From Existing Sources >> Code >> Unmark All >> Select Top Folder >> Ok
  - Подключить модуль к проекту project1
  - Project Structure >> Module >> Select project1 >> Dependencies >> + >> Module >> Select code >> ok
  - Теперь путь скажем к файлу ../code/net/MyClass.java >> import net.MyClass

## Конфликт имен

- при подключении библиотек с одинаковыми классами возникает конфликт имен
    - для разрешения конфликта надо оставить одну библиотеку, а классы второй указать напрямую
    - Пример
- ```
import java.util.Vector;           // конфликт имен
import net.mindview.simple.Vector; // конфликт имен

public class Ex2 {
    public static void main(String[] args) {
        Vector v = new Vector();
        Vector vector = new Vector();
    }
}
```
- Решение
- ```
import net.mindview.simple.List;
import net.mindview.simple.Vector; // конфликт имен

public class Ex2 {
    public static void main(String[] args) {
        Vector v = new Vector();
        java.util.Vector vector = new java.util.Vector();
```

## Пользовательские библиотеки

- Пользовательская библиотека исходников позволяет подменить стандартные функции
- Создание библиотеки
  - создать библиотеку классов в виде файлов .java
  - скопировать все файлы в папку code/lib/\*.java
- Подключение библиотеки в проекте IDEA
  - в проекте создать модуль из существующих исходников > code/\*
  - в структуре проекта подключить модуль в разделе dependencies > code
- Подключение библиотеки в файле проекте
  - на уровне класса директива > import lib/class;
  - на уровне статических функций класса > import static lib/class.\*;
  - > import static lib/class/method;

## Условная компиляция

- Условная компиляция не поддерживается в Java
- Метод условной компиляции при помощи директивы import
  - суть блокировани/ разблокирования импорта одноименных классов и методов
  - Пример
- //import static libtest.DebugOff.debug; //импорт метода debug() который работает как Off  
import static libtest.Debug.debug; //импорт метода debug() который работает как On

```
public class Ex1 {
    public static void main(String[] args) {
        debug();                                // активный метод debug() который работает как On
```

## Статический импорт

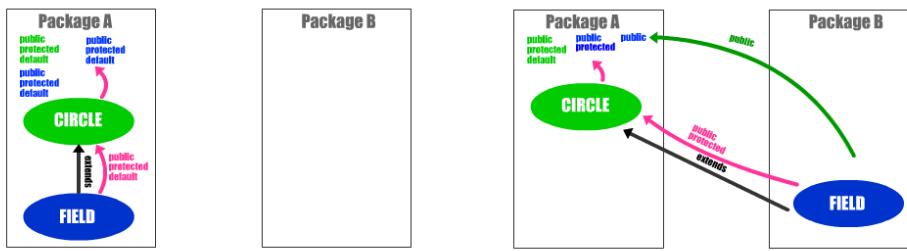
- Импорт на уровне функций
  - import static Class.\* означает импорт на уровне статических методов, их можно вызвать напрямую
  - import static net.mindview.util.Print.\*;

## Инкапсуляция

- Инкапсуляция это сокрытие данных и методов через ограничение доступа
- существует четыре уровня доступа private, default, protected, public

## Спецификаторы Доступа

- Спецификаторы доступа это private, default, protected, public
- Четыре варианта доступа
  - private              внутри класса
  - default             только в рамках пакета
  - protected          только в рамках пакета, но по каналу subclass отовсюду
  - public               отовсюду



## Реализация доступа к защищенным данным и методам

- Доступ к защищенным данным всегда и только через методы класса
  - private              только через методы класса
  - default             только через методы класса для другого пакета аналогично private
  - protected          через методы класса потомка или методы класса предка

## Реализация доступа к защищенным классам

- Классы могут быть только default или public
  - классы не public недоступны и невидимы вне библиотеки
  - классы public доступны вне библиотеки
- Организация классов в библиотеке
  - каждый открытый класс имеет свой файл с таким же именем
  - в файле может быть только один public класс и сколько угодно non-public
  - в файле может быть ни одного public класса
- Доступ к классам
  - невидимые классы доступны из классов и методов внутри package
  - все закрытые поля доступны из методов закрытых классов
- Полное закрытие класса
- Для полного закрытия класса
  - объявить класс non public
  - объявить все его конструкторы private
  - создать public static метод класса который будет возвращать экземпляр класса
  -

[lesson\\_ch06/ch/ex7](#)

## Разделение Интерфейса и Реализации

- Организация разделения интерфейса и реализации
  - это нужно, чтобы менять реализацию методов, не меняя интерфейса класса доступного клиенту
- Методика разделения
  - назначить public к данным, методам и классам которые будут открыты клиентам
  - назначить non public к данным, методам и классам которые будут доступны только разработчикам
  - разместить в коде сначала public данные и методы, затем скрытые данные и методы

## Повторное использование классов

- Есть два вида использования классов композиция и наследование

## Композиция

- Композиция или агрегация это встраивание классов внутрь других классов
  - при композиции в теле существующего класса находятся экземпляры других классов
  - все методы и данные экземпляров встроенных классов доступны в их исходном виде
  - модификаторы доступа работают также, если встроить класс из другого пакета
  - можно и получить серьезные ограничения в доступе к методам и данным класса
- Синтаксис
  - внутри класса создаются примитивы и ссылки на объекты других классов

## Инициализация при Композиции

- Порядок инициализации
  - сначала вызываются конструкторы встроенных классов
  - затем вызывается конструктор текущего класса
- Инициализация примитивов
  - может быть как в строке объявления, так и конструкторе
- Инициализация объектов
  - в точке объявления проводится конструктором
  - в конструкторе класса
  - отложенная инициализация непосредственно перед использованием н-р после проверки на null
  - в блоке инициализации {}

```
public class Bath {  
    private String  
        s1 = "Happy",  
        s2,  
        s3;  
    private Soap soap;  
    private int i;  
    private float toy;  
    public Bath() {  
        s2 = "Glad";           // инициализация в конструкторе  
        soap = new Soap();    // инициализация в конструкторе  
    }  
    {  
        i = 12;              // анонимный блок инициализации  
        toy = 2.34f;          // анонимный блок инициализации  
    }  
    @Override  
    public String toString() {  
        if (s3 == null) {  
            s3 = "Smile"; // инициализация отложенная в методе  
        }  
        return ("s1 = " + s1 + "\n");  
    }  
}
```

## Наследование

- Наследование это когда данные и методы предка переходят в состав данных и методов потомка
  - при наследовании можно в теле предка создавать классы потомков тогда методы будут потомков
- Синтаксис
  - наследование задает директива extends
  - для доступа к методам и данным предка из тела потомка используется ссылка super
- Super
  - это ссылка на экземпляр класса предка в теле класса потомка
  - по ссылке super можно получить доступ к всем public, protected, default данным и методам предка

## Инициализация при Наследовании

- Инициализация классов при наследовании происходит автоматически
  - при создании экземпляра класса потомка последовательно вызываются конструкторы
  - первым идет от самого дальнего предка до потомка, завершается конструктором потомка
- Инициализация классов при наследовании с вложенными классами
  - конструкторы встроенных в класс самого первого предка классов <встроенные>
  - конструктор самого первого предка классов
  - конструкторы классов предков, встроенные классы, до текущего класса потомка
  - конструкторы классов встроенных в текущий класс потомка
  - конструктор потомка
- Инициализация с Параметрами
  - параметры передаются по цепочке от потомка к предку с использованием super()
  - super(parameter) должен быть первым в объявлении конструктора потомка

```
public ClassC(int i) {
    super(i); // обязательно в первой строке
    System.out.println("ClassC Constructor with Integer");
    this.i = i;
}
public ClassC() {
    super(1); // обязательно в первой строке
    System.out.println("ClassC Constructor");
}
```

## Скрытие имен

- Перегруженные методы при наследовании не скрываются
  - перегрузка метода у потомка НЕ СКРЫВАЕТ все версии перегруженного метода предка
  - в отличие от других языков, все перегрузки метода предка доступны потомку

```
public class Homer {
    public char doh(char c) {
        System.out.println("doh(char): "+c);
        return 'd';
    }
    public float doh(float f) {
        System.out.println("doh(float): "+f);
        return 1.0f;
    }
}
public class Bart extends Homer {
    public void doh(MilHouse m) {
        System.out.println("doh(Milhouse m): "+m); // параметр ссылка на объект
    }
}
public static void main(String[] args) {
    Bart bart = new Bart();
    bart.doh(new MilHouse());
    bart.doh('c'); // char
    bart.doh(2.71f); // float
}
```

## Делегирование

- Делегирование это переопределение метода без внесения изменений
  - по сути это просто заключение метода одного класса в обертку другого класса

```
public class Cleanser { // класс методы которого будут делегированы
    private String s = "Cleanser # ";
    void append(String s) {
        this.s += s;
    }
    void scrub() {
        append("scrub() # ");
    }
}
public class Detergent{ // класс который будет делегировать методы другого класса
    private Cleanser cleanser = new Cleanser();
    public void append(String string) { // делегированный метод
        cleanser.append(string);
    }
    public void scrub() { // делегированный метод
        cleanser.scrub();
    }
}
```

## Композиция и Наследование

- Сочетание композиции и наследования

- применяется часто для создания комбинированных классов

```
public class Spoon extends Utensil { // класс Spoon наследует Utensil
    public Spoon(int i) {
        super(i);
    }
}
public class DinnerPlate extends Plate{ // класс DinnerPlate наследует Plate
    public DinnerPlate(int i) {
        super(i);
    }
}
public class PlaceSetting extends Custom{ // класс PlaceSetting наследует Custom
    private Spoon sp; // PlaceSetting комбинированный класс
    private DinnerPlate dp; // наследует Custom встраивает посуду
    public PlaceSetting(int i) {
        super(i+1); // класс предка Custom
        sp = new Spoon(i+2); // встроенный класс содержит предка
        dp = new DinnerPlate(i+5); // встроенный класс содержит предка
    }
}
```

## Композиция в сравнении с Наследованием

- Использование Композиции
  - при композиции все встроенные классы объявляются private
  - встроенные классы предоставляют внешнему классу свои поля и методы, то есть функционал
  - интерфейс с миром внешнего класса остается неизменным, но подключен к новому функционалу
  - Композиция использует Функционал встроенных классов, Интерфейс остается неизменный
  - Композиция отвечает на вопрос «имеет ли что то внутри себя»
- Использование Наследования
  - при наследовании внешний класс потомка является модификацией класса предка
  - интерфейс с внешним миром определяется интерфейсом предка
  - Наследование использует Интерфейс потомка, который модификация Интерфейса класса предка
  - Наследование отвечает на вопрос «является ли чем то»

## **Ключевой вопрос когда что выбирать Композицию или Наследование**

- **ВНИМАНИЕ.** Применять Композицию чаще, так как она лучше вписывается в концепцию ООП
- Применять Наследование обязательно если будет применено Восходящее преобразование
- Композиция для повторного использования реализации в новом классе при неизменном интерфейсе
  - композиция обеспечивает большую гибкость
  - может, используя наследование, менять тип, поведение встроенных объектов во время выполнения
  - отвечает на вопрос «новый класс имеет ли что то»
- Наследование для повторного использования интерфейса базового класса в новом классе
  - позволяет применить восходящее преобразование, что очень важно при полиморфизме
  - отвечает на вопрос «новый класс является ли чем то»

## **Общий смысл проектирования**

- Описание метода
  - при построении системы вся система разбивается на объекты и строится иерархия объектов
  - каждый класс строится не слишком крупным, не слишком малым
  - если архитектура слишком тяжелая, в нее вносятся новые классы, которые дробят объекты на части
  - наследование и композиция позволяют решать такие вопросы

## Завершение программы при Наследовании или Композиции

- Организация правильного завершения
  - Создать свой метод завершающих действий, тщательно его проработать
  - **ВНИМАНИЕ.** Не полагаться на уборщик мусора и метод finalize
  - **ВНИМАНИЕ.** Использовать секцию finally, которая выполняется ВСЕГДА, до закрытия программы
  - Приоритет перед finally есть только у команды System.exit(0)

## Наследование завершение

- Порядок создания и завершения при Наследовании
  - конструкторы сначала конструктор предка > затем конструктор потомка
  - конструктор потомка должен в первой строке вызвать конструктор super(params)
  - завершение сначала завершение потомка > затем завершение предка
  - метод завершения потомка должен в последней строке вызвать завершение super(params)

## Композиция завершение

- Порядок создания и завершения при Композиции
  - конструкторы конструктор класса > внутри конструкторы встроенных классов в прямом порядке завершение завершение класса > внутри завершение встроенных классов в обратном порядке
  - Пример
  - используется метод пользователя внутри класса, задача метода правильно закрыть объект класса
  - при наследовании методы вызываются вплоть до самого верхнего предка
  - при композиции закрытие идет в обратном порядке по сравнению с открытием
  -

```
public class Shape {  
    public Shape(int i) {  
    }  
    void dispose() {  
    }  
}  
public class Circle extends Shape {  
    public Circle(int i) {  
        super(i);  
    }  
    void dispose() {  
        super.dispose();  
    }  
}  
public class Triangle extends Shape {  
    public Triangle(int i) {  
        super(i);  
    }  
    void dispose() {  
        super.dispose();  
    }  
}  
public class CadSystem extends Shape {  
    private Circle c;  
    private Triangle t;  
    public CadSystem(int i) {  
        super(i+1);  
        c = new Circle(1);  
        t = new Triangle(1);  
    }  
    public void dispose() {  
        t.dispose(); // в обратном порядке  
        c.dispose(); //  
        super.dispose(); // базовый класс для CadSystem  
    }  
}
```

## Пример завершения смешанного класса

- Пример

```
public class Root {                                // содержит встроенные классы Component
    int[] ints;
    Component1 component1;
    Component2 component2;
    Component3 component3;
    public Root(int[] ints) {
        component1 = new Component1(ints[0]);      // первый создается
        component2 = new Component2(ints[1]);      // второй создается
        component3 = new Component3(ints[2]);      // третий создается
        System.out.println("Root.Constructor Done"); // общие данные создаются последними
    }
    void dispose() {
        System.out.println("Root.dispose"); // общие данные удаляются первыми
        component3.dispose(); // первым удаляется
        component2.dispose(); // вторым удаляется
        component1.dispose(); // третьим удаляется
    }
}
public class Stem extends Root {                  // наследует Root
    public Stem(int[] ints) {
        super(ints);                          // создание предка в первой строке
        System.out.println("Stem.Constructor ints[]"); // в конце операции по созданию потомка
    }
    public void dispose() {
        System.out.println("Stem.dispose ints[]"); // сначала операции по завершению потомка
        super.dispose();                      // завершение предка
    }
}
```

## Восходящее преобразование типов

- Суть восходящего преобразования
  - размещение объекта потомка в ссылке типа предка называется восходящим преобразованием
  - восходящее потому что предок расположен выше потомка в иерархии классов наследования
  - это также ведет к полиморфизму, изменению функционала переопределенных методов предка
  - **ВНИМАНИЕ.** Из объекта предка нельзя обратиться к полям и методам потомка, хотя они есть

```
public class Wind extends Instrument {
    public void play() {
        System.out.println("Wind.play");
    }
}
public static void main(String[] args) {
    Instrument flute = new Wind();
    flute.play();
}
```

## Нисходящее преобразование

- Суть нисходящего преобразования в расширении от предка к потомку [lesson\\_ch09/ch/ex18/downstream](#)
  - требуется явный кастинг, компиляции работает, на уровне Runtime не работает
  - ВНИМАНИЕ.** На уровне Runtime работает если кастинг с потомком, с которым создавался предок
  - Или если вызывается метод потомка, а реальный объект ближе к предку
  - Пример. Rodent >> Mouse >> FieldMouse
- Восходящее преобразование
  - когда потомок в теле предка
- Нисходящее преобразование
  - Когда объект потомка в теле предка затем принудительно проходит кастинг на объект потомка
  - Если объект реального потомка и кастинга совпадают, методы работают
  - ВНИМАНИЕ.** Если реальный потомок и кастинг не совпадают, ошибка вылетает только в Runtime
  -
- ```
public class Rodent {  
    public void eat() {  
        System.out.println("Rodent.eat");  
    }  
    public void sleep() {  
        System.out.println("Rodent.sleep");  
    }  
}  
public class Mouse extends Rodent{  
    public void mouseType() {  
        System.out.println("Mouse.type");  
    }  
}  
public class FieldMouse extends Mouse {  
    public void mouseColor() {  
        System.out.println("FieldMouse.color");  
    }  
}  
public class Ex18 {  
    public static void main(String[] args) {  
        System.out.println("\nExercise Downstream Class Check\n");  
        Rodent fieldMouse = new FieldMouse();  
        Rodent mouse = new Mouse();  
        fieldMouse.eat(); // восходящее преобразование  
        mouse.sleep(); // восходящее преобразование  
        ((Mouse)mouse).mouseType(); // нисходящее преобразование  
        ((FieldMouse)mouse).mouseColor(); // не сработает в Runtime  
        ((FieldMouse)fieldMouse).mouseColor(); // нисходящее преобразование  
    }  
}
```

## Полиморфизм

- Полиморфизм это размещение объекта потомка по ссылке класса предка
  - в этом случае интерфейс по ссылке предка будет точно равен интерфейсу предка
  - переопределенные методы будут методы потомка
  - использование методов потомка по ссылке предка возможно благодаря позднему связыванию
- Позднее связывание
  - это определение метода который будет выполняться на этапе выполнения кода
  - позднее связывание работает для всех методов, кроме private и final
- Расширяемость
  - полиморфизм параметров метода позволяет не меняя типа параметра расширить число типов
  - Пример весь оркестр работает через полиморфизм, в итоге вызываются методы инструментов
- Универсальность (мое определение)
  - независимо от вложенности методов в базовом классе сработают всегда те, которые переопределены независимо от того, на каком уровне вложенности они сработали

```
public class Instrument {  
    public void play(Note note) {           // метод предка который переопределен в потомках  
    }  
}  
public class Wind extends Instrument {  
    @Override  
    public void play(Note note) {           // переопределение метода предка  
    }  
    @Override  
}  
public class Music {  
    public static void tune(Instrument instrument) {  
        instrument.play(Note.MIDDLE_C); // вызывает метод потомка в теле предка  
    }  
    public static void tuneAll(Instrument[] instruments) {  
        for (Instrument instrument : instruments) {  
            tune(instrument);  
        }  
    }  
}  
public static void main(String[] args) {  
    Instrument[] orchestra = new Instrument[] { // массив Instrument предка для потомков  
        new Wind(),  
        new Percussion(),  
        new Stringed(),  
        new Brass(),  
        new WoodWind()  
    };  
    Music.tuneAll(orchestra);  
}
```

## Проблемы наследования

- Проблема переопределения закрытых методов
  - `private` методы класса предка скрыты от объекта ссылки предка
  - поэтому метод потомка **НОВЫЙ** и предок его просто не узнает, не прокатит даже компиляция
  - **ВНИМАНИЕ.** Переопределить можно только доступные методы которые видят объект предка
  - **РЕШЕНИЕ.** Методы `private` предка и `public` потомка должны иметь **РАЗНЫЕ** имена

```
public class PrivateOverride {  
    private void f() {  
    }  
    private void getOverrided() {  
    }  
}  
  
public class Derived extends PrivateOverride {  
    public void f() {  
    }  
    public void getOverrided() {  
    }  
}  
  
public class Ex11 {  
    public static void main(String[] args) {  
        PrivateOverride prOver = new Derived();  
        Derived derived = new Derived();  
        //      privateOverride.getOverrided(); // не работает, потому что экземпляр parent  
        //      не видит свой getOverrided  
        // и для него это новый метод потомка о  
        //котором он ничего не знает  
        //не работает это новый метод о котором  
        //экземпляр предка ничего не знает  
        derived.getOverrided();  
        derived.f();  
        // работает это метод потомка  
        // работает это метод потомка  
    }  
}
```

- Проблема полей класса

- обычные поля класса **НЕ ПОДДЕРЖИВАЮТ** полиморфизм
- при прямом доступе из ссылки предка к полю класса доступ идет к полю предка

- `public class PrivateOverride {  
 public int iField = 12;  
 public int getField() {  
 return iField;  
 }  
}`
- `public class Derived extends PrivateOverride {  
 public int iField = 15;  
 @Override  
 public int getField() {  
 return iField;  
 }  
}`
- `public class Ex11 {  
 public static void main(String[] args) {  
 PrivateOverride prOver = new Derived();  
 Derived derived = new Derived();  
 System.out.println("derived.iField=" + prOver.iField); //не работает берет поле предка  
 System.out.println(prOver.getField()); //работает override метод берет поле потомка  
 }  
}`

- Проблема статических методов класса

- статические методы **НЕ ПОДДЕРЖИВАЮТ** полиморфизм
- для объекта потомка по ссылке предка вызываются статические методы предка

```
public class PrivateOverride {  
    public static void staticOverrided() {  
    }  
}  
  
public class Derived extends PrivateOverride {  
    public static void staticOverrided() {  
    }  
}  
  
public class Ex11 {  
    public static void main(String[] args) {  
        prOver.staticNotOverrided(); // не работает вызывает static методы предка  
        prOver.staticOverrided(); // не работает, вызывает static метод предка  
    }  
}
```

[lesson\\_ch08/ex11/Ex11](#)

[lesson\\_ch08/ex11/Ex11](#)

## Конструкторы и Полиморфизм

- Поведение конструкторов при полиморфизме
  - Конструкторы не поддерживают полиморфизм, так как являются статическими методами
  - порядок вызова конструкторов по нисходящей от предка к потомку
  - **ВНИМАНИЕ.** запомнить порядок создания полей, нужно при завершении в обратном порядке
  - **ВНИМАНИЕ.** сделать подсчет ссылок на совместно используемые объекты при создании

## Инициализация с полиморфизмом

- Общий порядок загрузки класса [lesson\\_ch8/ch/ex12](#)
  - класс хранится в отдельном файле и класс загружается в точке первого обращения к классу
  - обращение к классу либо через static поля, методы, либо через вызов конструктора класса
- Порядок инициализации класса с наследованием [ex11,ex11a](#) [lesson\\_ch08/ch/ex12](#)

|           |                     |                                          |
|-----------|---------------------|------------------------------------------|
| ○ предок  | static переменные в | месте объявления                         |
| ○ предок  | static переменные в | static блоке                             |
| ○ потомок | static переменные в | месте объявления                         |
| ○ потомок | static переменные в | static блоке                             |
| ○ предок  | переменные в        | месте объявления                         |
| ○ предок  | переменные в        | блоке                                    |
| ○ предок  | конструктор         | в первой строке можно разместить super() |
| ○ потомок | переменные в        | месте объявления                         |
| ○ потомок | переменные в        | блоке                                    |
| ○ потомок | конструктор         | в первой строке можно разместить super() |

## Завершение при полиморфизме

- Завершение должно проходить в обратном порядке [lesson\\_ch8/ch/ex12](#)
  - порядок вызова программ завершения dispose() по восходящей от потомка к предку
  - поля закрываются в обратном порядке по сравнению с открытием
  - сначала вызывается программа завершения потомка, затем программа завершения предка
  - **ВНИМАНИЕ.** переопределить завершение в потомке, вызвать последней строкой super.dispose
  - **ВНИМАНИЕ.** закрытие совместно используемых объектов производится через подсчет ссылок
- Завершение при композиции и полиморфизме [lesson\\_ch8/ch/ex12a](#)
  - если объект содержит массив ссылок и все они равны одному объекту другого класса
  - dispose() первой ссылки массива удалит другой объект, dispose() второй ссылки вызовет ошибку
  - **ВНИМАНИЕ.** для завершения при ссылках на один и тот же объект используется подсчет ссылок
  - **ВНИМАНИЕ.** наследование и полиморфизм не влияют на подсчет ссылок, так как потомок
  - содержит в себе всю вертикаль предков и его можно рассматривать как один объект
  - **ВНИМАНИЕ.** finalize() позволяет отследить потерянные ссылки [lesson\\_ch08/ex14](#)

```
public abstract class Singer {
    protected abstract void addRef();
}

public class Tenor extends Singer {
    private int refcount = 0;
    public Tenor() {
        super();
    }
    public void addRef() { // свой счетчик ссылок
        refcount++; // переопределять надо, у предка абстрактный метод
    }
    protected void dispose() {
        if (--refcount == 0) {
            super.dispose();
        }
    }
}
```

## Обработка референсных ссылок при полиморфизме

- Создание контроля референсных ссылок [lesson\\_ch08/ch/ex12a/ex14/ex15](#)
  - создать private int refcount = 0 в классе объекте и классе потомке при полиморфизме
  - создать addRef() { refcount++} в классе объекте и классе потомке
  - создать addref() {} в классе предка, пустой или абстрактный метод
  - вызвать object.addRef() в конструкторе внешнего класса именно как метод объекта
  - parent.addRef() в конструкторе внешнего класса, метода как метод предка
- Завершение работы объекта
  - создать код if(--refcount== 0) в методе dispose() класса объекта или класса потомка
  - создать код super.dispose() вызвать dispose() класса предка из dispose() потомка

## Finalize Check при полиморфизме

- Обработка finalize() при полиморфизме [lesson\\_ch08/ch/ex12a/ex14/ex15](#)
  - создать boolean check = false во всех конечных объектах и в объектах потомках
  - check = true в конструкторе класс объекта или класса потомка
  - check = false в dispose() класса объекта или класса потомка
  - if(check) в finalize() класс объекта или класса потомка

## Полиморфные методы при вызове из конструкторов

- Вызов переопределенного метода в конструкторе предка ведет к ошибке [lesson\\_ch08/ch/ex16](#)
  - потому что вызывается метод потомка, и данные потомка, которые еще не инициализированы
  - **ВНИМАНИЕ.** переопределенные методы в конструкторе предка работают с не готовыми данными
  - Пример

```
public class Glyph {  
  
    public Glyph() {  
        System.out.println("Glyph.Glyph.before.draw");  
        draw(); // rewritten method  
        System.out.println("Glyph.Glyph.after.draw");  
    }  
    void draw() {  
    }  
}  
public class RoundGlyph extends Glyph{  
    private int radius = 1;  
  
    public RoundGlyph(int radius) {  
        this.radius = radius;  
        System.out.println("RoundGlyph.Constructor.radius."+radius);  
    }  
    void draw() {  
        System.out.println("RoundGlyph.draw.radius."+radius);  
    }  
}
```

## Ковариантность

- Ковариантность возвращаемых типов [lesson\\_ch08/ch/ex17](#)
  - Переопределенный метод потомка может вернуть тип ПРОИЗВОДНЫЙ от типа предка
- Ковариантность типов входных параметров методов [lesson\\_ch09/ch/ex11/Downcase](#)
  - Переопределенный метод потомка должен иметь ТОЧНО ТАКОЙ же тип параметра как у предка

## Наследование при проектировании

- Наследование использовать только если композиция не подходит
- Композиция более гибкая
  - композиция позволяет на лету изменять тип объекта
  - при этом используется наследование для смены типа встроенных объектов
  - Пример
- ```
public class Stage {  
    private Actor actor = new HappyActor(); // изначально тип объекта HappyActor  
    public void change() {  
        actor = new SadActor();  
    }  
    public void performPlay() {  
        actor.act();  
    }  
}
```

## Нисходящее преобразование

- Динамическое определение типа

[lesson\\_ch08/ch/ex18](#)

- при полиморфизме компилятор не пропускает методы потомка по ссылке на объект предка
  - это происходит потому, что работает RTTI динамическое определение типа реального потомка

- Пример

```
public class Useful {  
    public void f() {  
    }  
}  
public class Math extends Useful {  
    public void f(){ // переопределенный метод  
    };  
    public void h(){ // новый метод  
    };  
}  
public static void main(String[] args) {  
    System.out.println("Dynamic Child Define Check");  
    Useful[] x = new Useful[]{ new Useful(), new Math() };  
    x[0].f(); // работает переопределенный метод  
// x[1].i(); // не может определить на этапе компиляции
```

# Интерфейсы

## Абстрактные классы и методы

- Абстрактный метод
  - это метод без реализации, может быть только у абстрактного класса
  - реализация метода обычно у потомков абстрактного класса
- Абстрактный класс это класс который не может иметь экземпляров
  - часть методов или даже все методы могут быть абстрактными может быть без реализации
  - абстрактный класс это класс предка, реализация методов должна быть у всех потомков
  - **ВНИМАНИЕ.** Полиморфизм работает, можно в теле абстрактного предка создать объект потомка

```
public abstract class Instrument {                                // абстрактный класс
    public Instrument() {                                         // конструктор
        System.out.println("Instrument.Constructor");
    }
    protected void dispose() {                                     // завершение работы класса
        System.out.println("Instrument.dispose");
    }
    public abstract void play(Note note);                         // абстрактный метод
    public abstract void adjust();                                // абстрактный метод

    @Override
    public String toString() {                                     // обычный метод
        return "Instrument{}";
    }
}

{
    ◦ ВНИМАНИЕ. Анонимный класс выглядит как экземпляр абстрактного класса но нет, это потомок
    ◦ ВНИМАНИЕ. Анонимный класс имеет видимость своей this только ВНУТРИ своих методов
}

public static void main(String[] args) {
//    Limpopo limpopo = new Limpopo(); // не работает, если не создавать анонимный класс
    Limpopo limpopo = new Limpopo() { // экземпляр анонимного класса потомка Limpopo
        String s1 = "St1";           // увидеть только из переопределенных методов
        int i = 15;                 // можно только из переопределенных методов
        void play() {               // переопределенный метод
            System.out.print(this.s1+" ");
            System.out.println("Anonimous.play");
        }
    };
    limpopo.play();
    limpopo.grow();
}
```

## Разница между методом тип static и abstract при наследовании

- Разницы между методами только в способе вызова методов
  - при получении полиморфного объекта оба метода распознают потомка, результат одинаковый
  - статический метод вызывается как метод класса предка
  - реализация абстрактного метода вызывается как метод потомка
- ```
abstract public class Stone {
    static void getBase(Stone stone) {                               // static метод класса предка
        System.out.println("Stone.getBase static > " + stone);
    }
    abstract void getBase2(Stone stone);                            // abstract метод предка
}
public class Tool extends Stone {                                  // реализация abstract метода потомка
    @Override
    void getBase2(Stone stone) {
        System.out.println("Tool.getBase2 > " + stone);
    }
}
```

## Интерфейсы

- Интерфейс это полностью абстрактный класс
  - интерфейс содержит только абстрактные методы по умолчанию (public)
  - интерфейс содержит только абстрактные поля, по умолчанию (public) (final) (static)
  - если интерфейс подключен к классу, класс обязан иметь реализации всех методов интерфейса
  - к классу может быть подключено любое число интерфейсов
  - **ВНИМАНИЕ.** Интерфейсы можно подключать на любом уровне при наследовании, вопрос гибкости
  - Пример

```
public interface Instrument {           // первый интерфейс
    int VALUE_INT = 5;                 // final public static
    void play(Note note);            // public void первый интерфейс
    String what();                   // analog toString
    void adjust();                  }

}

public interface Cargo {               // второй интерфейс
    void weight();                  // public void второй интерфейс
    void move();
}

public class Wind implements Instrument, Cargo{
    @Override // первый интерфейс Instrument
    public void play(Note note) {
    }
    @Override
    public String what() {
    }
    @Override
    public void adjust() {
    }
    @Override // второй интерфейс Cargo
    public void weight() {
    }
    @Override
    public void move() {
    }
    @Override
    public void check() {
    }
}
```

## Отделение интерфейса от реализации

- Сама концепция интерфейса это механизм отделения интерфейса класса от его реализации
  - это позволяет использовать один и тот же интерфейс с совершенно разными реализациями
- Пример на базе наследования [lesson\\_ch09/ch/ex11](#)
  - базовый класс Process, от него два потомка классы StringProcessor и FilterAdapter
  - StringProcessor разветвляется на подклассы обработки текста восходящее преобразование
  - FilterAdapter транслирует интерфейс в Filter, делегирование метода
  - Filter разветвляется на подклассы обработки звука восходящее преобразование
  - **ВНИМАНИЕ.** приведен только код базового класса, остальноекак в примере на базе интерфейса
  -

```
abstract public class Processor {
    abstract public Object getData();
    public String name() {
        return getClass().getSimpleName(); // get simple Name of Class
    }
    public Object process(Object object) { // фиктивный метод ковариантность выхода
        return object;
}
```

- Пример на базе интерфейса

[lesson\\_ch09/ch/ex11](#)

- базовый интерфейс IProcessor, от него два потомка классы StringProcessor и FilterAdapter
- StringProcessor разветвляется на подклассы обработки текста восходящее преобразование
- FilterAdapter транслирует интерфейс в Filter, делегирование метода
- Filter разветвляется на подклассы обработки звука восходящее преобразование

```

public interface IProcessor {
    public String name();
    public Object process(Object object);
    public Object getData();
}

abstract public class DStringProcessor implements IProcessor {
    public String s = "Test Case Local MNap";
    @Override
    public Object getData() {
        return s;
    }
    @Override
    public String name() {
        return getClass().getSimpleName(); // get simple Name of Class
    }
}
abstract public class DFilterAdapter implements IProcessor {
    public DWaveform w = new DWaveform();
    @Override
    public Object getData() {
        return w;
    }
    @Override
    public String name() {
        return getClass().getSimpleName(); // get simple Name of Class
    }
    @Override
    public Object process(Object object) {
        return process((DWaveform) object); // делегирование метода
    }
    abstract protected DWaveform process(DWaveform object); // переопределен восходящее
}

```

## Множественное наследование в Java

- Интерфейсы позволяют множественное наследование
  - это возможно потому, что интерфейсы НЕ содержат реализации поэтому не могут конфликтовать
  - в ромбовидном наследовании одинаковые методы не конфликтуют так как не реализованы
  -

```

public interface Insect { // базовый интерфейс
    void move();
}

public interface Fly {
    void attack(); // дублирует метод Poison
}

public interface Poison extends Insect {
    void attack(); // дублирует метод Fly
}

public class Bee implements Poison,Fly{
    @Override
    public void attack() {
        System.out.println("Bee.attack");
    }
    @Override
    public void move() {
    }
}

```

## Наследование интерфейса

- Расширение интерфейса через наследование
  - интерфейсы допускают наследование, позволяют расширять интерфейс за счет наследования

## Конфликты при совмещении интерфейсов

- Конфликт возникает если одноименная функция возвращает разные типы в

- при наследовании в базовом классе и базовом интерфейсе
  - при наследовании интерфейсов в разных базовых интерфейсах

- Конфликт по наследованию в базовом классе и интерфейсе
  -

```
public class C {
    int f() {                                // функция возвращает int
        return 1;
    }
}
public interface I1 {
    void f();                                // функция возвращает void
}
public class C5 extends C implements I1 { // конфликт C{int f()} и I1 { void f()}
@Override
    public void f() {                        // конфликт возвращаемого значения
    }
```

- Конфликт по наследованию в базовых интерфейсах

```
public interface I1 {
    void f();                                // функция возвращает void
}
public interface I3 {
    int f();                                 // функция возвращает int
}
public interface I4 extends I1, I3 {         // конфликт I1 {int f()} и I3 { void f()}
}
```

## Интерфейсы как средство адаптации

- Механизм интерфейсов позволяет создавать разные реализации одного и того же интерфейса
- Порядок работы с интерфейсом
  - подключить объект к интерфейсу и сделать реализацию методов интерфейса
  - теперь этот объект может быть представлен в любой системе как потомок интерфейса
  - Пример. Интерфейс Readable (, реализован метод read()) позволяет объекту работать с Scanner()

- ```
public class RandomWords implements Readable {
    final int LEN_OF_WORD = 15;
    private Random rnd = new Random();           // получили объект
    private int count = 8;                         // NUM_OF_WORD;
    @Override
    public int read(CharBuffer cb) throws IOException { // заполнить буфер, вернуть число
        if (count-- == 0) {                         // конец входных данных по read
            return -1;
        }
        for (int i = 0; i < LEN_OF_WORD ; i++) {
            cb.append( Integer.toString(rnd.nextInt(10)) ); // цифру в символ
        }
        cb.append(" ");
        return LEN_OF_WORD+1;                         // разделитель " " между словами
    }                                              // длина символов массива + " "
    }
}
public static void main(String[] args) {
    Scanner in = new Scanner(new RandomWords());
    while (in.hasNext()) {
        System.out.println(in.next());
    }
}
```

## Интерфейс и шаблон Адаптер

- Использование Адаптера для подключения интерфейса к классу с неизменным интерфейсом
    - класс с неизменным интерфейсом взять как базовый или встроенный для класса адаптера
    - класс адаптера подключить к нужному интерфейсу и использовать как внешний класс для работы
- ```
public class CharRandom {           // класс с неизменным интерфейсом, генерит 12 символов
    private final char[] chs =
        new String("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
    private final int NUM_OF_CHAR = 12;
    private char[] c = new char[NUM_OF_CHAR]; // массив для символов
    private Random rnd = new Random();

    char[] genChars() {
        for (int i = 0; i < NUM_OF_CHAR; i++) {
            c[i] = chs[rnd.nextInt(chs.length)];
        }
        return c;
    }
}

public class CharAdapter implements Readable{ // класс адаптер с Readable интерфейсом
    CharRandom charRandom = new CharRandom();
    private final int NUM_OF_WORDS = 10;
    private int count = NUM_OF_WORDS;
    @Override
    public int read(CharBuffer cb) throws IOException { // метод read() для Scanner
        if (count-- == 0) {
            return -1;
        }
        String s = String.valueOf(charRandom.genChars()) + " ";
        cb.append(s);                                // символы в строку
        return s.length();
    }
}

public class Ex16 {
    public static void main(String[] args) {
        System.out.println("\nExercise 16 Char Readable Interface Adapter Check\n");
        Scanner pass = new Scanner(new CharAdapter()); // генерация 8 слов по 12 символов
        GenClass.show(pass);
    }
}
```

## Поля в интерфейсах

- Поля в интерфейсах по умолчанию public final static lesson\_ch09/ch/ex17
  - даже если не показано, все равно public static final и это не меняется
  - раньше использовались для реализации enum()
- Инициализация полей интерфейса
  - поля можно инициализировать константой или Random() или любой функцией
  - поля инициализируются один раз за запуск программы, потому что static

## Вложенные интерфейсы

- Вложенные интерфейсы это интерфейсы вложенные в класс или интерфейс [lesson\\_ch09/ch/ex18](#)
- Вложенные классы
  - это класс объявленный внутри другого класса
  - может быть использован ТОЛЬКО ВНУТРИ класса или его потомков, где объявлен
  - имеет все уровни доступа private, default, protected, public
- Вложенные интерфейсы внутри класса [lesson\\_ch09/ch/ex18](#)
  - работает как независимый интерфейс, адресация ClassName.IName
  - может быть private, default, protected и public все работает
  - может быть использован извне без проблем в пределах уровня доступа
- Вложенные интерфейсы внутри интерфейса [lesson\\_ch09/ch/ex18](#)
  - работает как независимый интерфейс, адресация IName.IName
  - может быть только public
  - может быть использован извне без проблем
- **ВНИМАНИЕ.** Ограничения на вложенные классы это использование внутри класса или его потомков  
Ограничения на вложенные интерфейсы это уровень доступа

## Интерфейсы и шаблон Фабрика

- Шаблон фабрика работает так на примере Game [lesson\\_ch09/ch/ex18/gamer /ex19/](#)
  - создается несколько реализаций одного интерфейса IGame, которые реализуют саму игру
  - создается несколько реализаций одного интерфейса IGetGame, фабрики игры создают экземпляр
  - затем вызывается метод играть, в качестве параметра экземпляр игры в интерфейс IGetGame
  - Итого. Метод playGame позволяет создать любую игру, которая поддерживает интерфейс IGetGame
  - Пример

```
public interface Game { // интерфейс самой игры
    boolean move();
}

public class Checkers implements Game {
    private final int MOVES_CNT = 3;
    private int moves = 0;

    @Override
    public boolean move() {
        System.out.println("Checkers.move." + ++moves);
        return (moves < MOVES_CNT);
    }
}

public interface GameFactory { // интерфейс фабрики игры которая создает экземпляр
    Game getGame();
}

public class CheckersFactory implements GameFactory{ // реализация фабрики игры
    @Override
    public Game getGame() {
        return new Checkers(); // выдать экземпляр игры в шашки
    }
}

public class PlayGame { // статический метод играть в игру
    public static void playGame(GameFactory gameFactory) {
        Game game = gameFactory.getGame(); // получить игру из интерфейса игр
        while (game.move()) {
        }
    }
}
```

## Внутренние классы

- Внутренний класс это класс который размещен прямо внутри класса
  - внутренний класс скрывает код, ExtClass.внешний класс, InnerClass.внутренний класс
- **ВНИМАНИЕ.** InnerClass имеет доступ к закрытым полям и методам ExtClass через ExtClass.this
- **ВНИМАНИЕ.** ExtClass имеет доступ к закрытым полям и методам InnerClass через экземпляр InnerClass
- Создание объектов внутреннего класса
  - Способ 1      используется метод внутри внешнего класса
  - Способ 2      использовать оператор .new
  - Способ 3      использовать наследование

```
Parcel1 p1 = new Parcel1(); // получить ссылку на внешний класс
Parcel1.Contents plc = p1.getContents(); // получить ссылку на внутренний класс
lesson_ch10/ch/ex1

lesson_ch10/ch/ex2

Sequence ss = new Sequence(10);
Sequence.Selector s2 = ss.new Selector(); // короткий синтаксис внутреннего класса
lesson_ch10/ch/ex6

public class Parcel4 {
    private class PContents implements IContents { // внутренний класс с интерфейсом
        @Override
        public int getValue() {
        }
    }
    public class Processing extends Fund {
        public IAccount getIAccount() {
            return new Fund.Account(); // прокатило есть наследование, предок четко увязан
        }
    }
    public class Destination { // inner class
        private String label;
        public Destination(String label) {
            this.label = label;
        }
        public String getLabel() {
            return label;
        }
        public Contents getContents() {
            return new Contents();
        }
        public Destination getDestination(String label) {
            return new Destination(label);
        }
        public void ship(String dest) {
            Contents c = new Contents(); // используем внутренний класс как обычный
            Destination d = new Destination(dest); // тоже самое, как обычный внешний класс
            System.out.println(d.getLabel() + ": " + c.getValue()); // распечатать Label
        }
    }
}
public class Ex1 {
    public static void main(String[] args) {
        System.out.println("\nInner Class Check\n");
        Parcel1 p1 = new Parcel1(); // получить ссылку на внешний класс
        p1.ship("Tanzania");

        System.out.println("\nInner Class Check\n");
        Parcel1.Contents plc = p1.getContents(); // получить ссылку на внутренний класс
        Parcel1.Destination pld = p1.getDestination("Paris");
        System.out.println(pld.getLabel() + ": " + plc.getValue());
    }
}
```

## Ссылка на внешний класс из внутреннего

- Внутренний класс получает неявно ссылку на внешний класс, который его создал
  - Поэтому внутренний класс может спокойно обращаться к полям и методам внешнего класса
  - Может получить ссылку на экземпляр класса который его создал
  - **ВНИМАНИЕ.** Ссылка на внешний класс из внутреннего через оператор `ExtClass.this`
  - Ссылка на внутренний класс из внутреннего класса `.this` или `InnerClass.this`
  - Пример
- ```
public class Sequence {  
    private Object[] items; // ссылка пустая на массив ссылок Object  
    private int next = 0;  
    private Sequence pLink = this; // ссылка на экземпляр внешнего класса  
    public class Selector implements ISelector { // inner class  
        private int index = 0;  
        @Override  
        public boolean checkEnd() { // проверяет последний это элемент или нет  
            return (index == items.length);  
        }  
        @Override  
        public Object getCurrent() { // вытаскивает текущий объект  
            if (checkEnd()) return null; // не даем упороть индекс  
            return items[index];  
        }  
        @Override  
        public void nextIndex() {  
            if (index < items.length) {  
                index++;  
            }  
        }  
        public void showLinks() {  
            System.out.println("Inner class link:" + this);  
            System.out.println("Inner class link:" + Selector.this); // внутренний класс  
            System.out.println("Ext Class link :" + pLink); // экземпляр внешнего класса  
            System.out.println("Ext Class link :" + Sequence.this); // тоже самое но короче  
        }  
    }  
    public Sequence(int size) {  
        items = new Object[size]; // создали массив пустых ссылок  
    }  
    public void add(Object o) {  
        if (next < items.length) {  
            items[next++] = o; // прописали новую ссылку в массиве на объект  
        }  
    }  
    public void showLinks() {  
        System.out.println("Ext Class link :" + this); // экземпляр внешнего класса  
    }  
    public Selector getInner() { // получить экземпляр внутреннего класса  
        return new Selector();  
    }  
}
```

## Операторы this и new

- Оператор `this` позволяет получить доступ из внутреннего класса к внешнему [lesson\\_ch10/ch/ex2/ex5](#)
    - синтаксис `ExtClassName.this` ссылка на экземпляр внешнего класса показано выше
  - Оператор `new` позволяет создать экземпляр внутреннего класса без метода внешнего класса
    - синтаксис `ExtClass.InnerClass inner = ext.new InnerClass()`
    - **ВНИМАНИЕ.** Можно в одну строку `ExtClass.InnerClass inner = new ExtClass().new InnerClass()`
    -
- ```
public static void main(String[] args) {  
    Sequence ss = new Sequence(10);  
    Sequence.Selector s2 = ss.new Selector(); // короткий синтаксис внутреннего класса  
}
```

## Внутренние классы восходящее преобразование

- Применение внутренних классов для реализации интерфейса
  - позволяет вытащить объект внутреннего класса наружу как ссылку с типом интерфейса
  - при этом даже имя класса и тип полностью скрыты, так как это закрытый внутренний класс
  - **ВНИМАНИЕ.** этот метод позволяет ПОЛНОСТЬЮ скрыть реализацию под ИНТЕРФЕЙСОМ
  - **ВНИМАНИЕ.** При наследовании вызов конструктора идет через точку `IFace ic = ExtClass.Inner();`
  - Пример

```
public class Parcel4 {  
    private class PContents implements IContents { //закрытый внутр. класс с интерфейсом  
        private int value = 11;  
        @Override  
        public int getValue() {  
            return value;  
        }  
    }  
    protected class PDestination implements IDestination { //закрытый внутренний класс  
        private String label;  
        PDestination(String label) {  
            this.label = label;  
        }  
        @Override  
        public String getLabel() {  
            return label;  
        }  
    }  
    public IContents getPContents() { // получить ссылку на объект внутреннего класса  
        return new PContents(); // тип ссылки равен интерфейсу  
    }  
    public IDestination getPDestination(String label) {  
        return new PDestination(label);  
    }  
}  
public static void main(String[] args) {  
    Parcel4 p4 = new Parcel4();  
    IContents pc = p4.getPContents(); // интерфейс внешний вытаскиваем ссылку на  
                                    //внутренний закрытый класс тип класса неизвестен  
    IDestination pd = p4.getPDestination("Toronto");  
    System.out.println(pd.getLabel()+":"+pc.getValue()); // работает как с обычной  
   // ссылкой по интерфейсу  
}
```

- **ВНИМАНИЕ.** Пример Кольцевого доступа loop access

[lesson\\_ch10/ch/ex7](#)

ExtMethod >> New Inner >> Inner.Method >> ExtMethod

```
public class Ground {  
    private int groundValue = 35; // закрытое поле внешнего класса  
    private class Router { // закрытый класс  
        private int groundValue = 15; // добавлено чтобы увеличить неразбериху  
        private void incGround() { // закрытый метод внутреннего класса  
            Ground.this.groundValue++; // доступ к закрытому полю внешнего класса  
            Ground.this.showGround(); // доступ к закрытому методу внешнего класса  
        }  
        private void showGround() { // тоже самое чтобы добавить неразберихи  
            System.out.println("Router:"+groundValue);  
        }  
        private void showGround() { // закрытый метод внешнего класса  
            System.out.println("Ground:"+groundValue);  
        }  
    }  
    public void newRouter() { // кольцевой доступ разрешено вызвать снаружи  
        Router r = new Router(); // создать объект закрытого внутреннего класса  
        r.incGround(); // вызвать его закрытый метод внутреннего класса  
    }  
}
```

## **Внутренние классы размещение в разных областях методах, классах и прочее**

- Использование может потребоваться в случаях
  - Когда есть возможность создавать ссылки через интерфейс пристегнутый к внутреннему классу
  - Когда есть нужда задействовать класс, но скрыть его реализацию полностью
- Области где можно создать и использовать внутренний класс [lesson\\_ch10/ch/ex8/ex9/ex10](#)
  - внутри метода
  - в области действия {} внутри метода if(){ область 1} else{ область 2}
  - анонимный класс, реализованный в интерфейсе ???
  - анонимный класс, внутри пользовательского конструктора класса ???
  - анонимный класс выполняющий инициализацию поля ???
  - анонимный класс выполняющий конструирование и инициализацию экземпляра
- Примеры. Смотреть в разделе HINTs

## Анонимные внутренние классы

- Анонимный внутренний класс это
  - потомок существующего класса с определением класса прямо в тексте метода
  - наследование распространяется на анонимный класс, методы наследуются, данные дублируются
  - инициализация полей в анонимном классе работает только с final
  - **ВНИМАНИЕ.** Анонимные классы это ВСЕГДА НАСЛЕДОВАНИЕ [lesson\\_ch10/ch/ex12/access](#)
  - **ВНИМАНИЕ.** Доступ к полям внешнего класса через имя класса как у вложенных классов Ext.this.
  - **ВНИМАНИЕ.** Доступ к полям предка как и всегда при наследовании через super.
- Создание анонимного класса на базе конструктора по умолчанию [lesson\\_ch10/ch/ex12/Parcel7](#)

```
public class Contents { // базовый класс основа анонимного класса
    private int value = 15;
    public Contents() {
    }
    public int getValue() {
        return value;
    }
}
public class Parcel7 {
    public Contents getContents() {
        return new Contents(){ // объявление анонимного класса через конструктор
            private int value = 17; // новое поле нового класса
            @Override // переопределенный
            public int getValue() { // метод возьмет поле анонимного класса
                return value;
            }
        }; // анонимный класс завершен
    }
}
```

- Создание анонимного класса на базе конструктора с параметрами [lesson\\_ch10/ch/ex12/Parcel8](#)

```
public class Wrapper {
    private int value;
    public Wrapper(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}
public class Parcel8 { // полная реализация анонимного класса
    public Wrapper getWrapper(int value) {
        return new Wrapper(){ // вызов анонимного класса
            @Override
            public int getValue() {
                return super.getValue()*17; // используется значение базового класса
            }
        }; // анонимный класс завершен
    } // функция завершена
}
```

- Создание анонимного класса и инициализация параметров анонимного класса [lesson\\_ch10/ch/ex12/](#)
  - возможно только с параметром типа final
  - Пример. Входное значение новое при каждом вызове метода, значит каждый экземпляр разный
  -

```

public class Parcel9 {
    public Contents getContents(final int intValue) {           // функция, любые параметры
        return new Contents() {                                     // конструктор
            private int value = intValue; // инициализируется при создании экземпляра
            @Override
            public int getValue() {           // выдать свое значение
                return value;
            }
        };
    }
}
Parcel9 p9 = new Parcel9();
System.out.println(p9.getContents(77).getValue()); // анонимный класс
System.out.println(p9.getContents(98).getValue()); // анонимный класс

```

- Создание имитации конструктора анонимного класса [lesson\\_ch10/ch/ex12/Parcel10](#)
  - анонимный класс не имеет конструктора, но его можно заменить блоком инициализации
  - приведена комбинированная система инициализации переменной и блоком
  -

```

public class Parcel10 {
    public Contents getContents(int intValue) {           // это объявление функции пользователя
        return new Contents() {                           // конструктор базового класса
            private int value = intValue;                 // final значение по любому

            {   // блок работает как конструктор
                int rnd1 = new Random().nextInt(100);
                int rnd2 = new Random().nextInt(100);

                System.out.println("Parcel10.instance initializer." + value + " " +
                    rnd1+" "+rnd2);

                if (rnd1 > rnd2) {                     // ставим наименьшее
                    rnd1 = rnd2;                      // такое может только конструктор
                }
                this.value += rnd1;
            }

            @Override
            public int getValue() {                  // выдать свое значение
                System.out.println("Parcel10.getValue");
                return value;
            }
        };
    }
}

```

- Доступ к членам внешнего класса и своего предка из анонимного класса      [lesson\\_ch10/ch/ex12/access](#)
  - к членам внешнего класса      через `имя.this.field`, `имя.this.method()`
  - к членам своего предка      через `super.field`, `super.method()`
- ```
public class Ground {
    private int groundValue = 35; // закрытое поле внешнего класса
    protected void showGround() { // закрытый метод внешнего класса
        System.out.println("Ground: " + groundValue);
    }
    protected void incGround(){ // абстрактная функция
    }
    protected void incValue(){ // абстрактная функция
    }
    public void newRouter() { // запустить отработку
        this.showGround(); // проверка содержимого поля базового класса
        getRouter().incGround();
        this.showGround(); // проверка содержимого поля базового класса
    }
    private Ground getRouter() {
        return new Ground() { // Anonym
            private int groundValue = 15; // anonym field
            @Override
            protected void incGround() {
                super.groundValue++; // доступ к полю super анонимного класса
                super.showGround(); // доступ к методу super анонимного класса
                Ground.this.groundValue++; // доступ к полю внешнего класса
                Ground.this.showGround(); // доступ к методу внешнего класса
            }
            @Override
            protected void showGround() {
                System.out.println("Ground:" +super.groundValue+ " Router:" + groundValue);
            }
            @Override
            protected void incValue() {
                groundValue++;
            }
        };
    }
};
```

## Генерация Анонимного класса из Интерфейса

- самый естественный путь создания анонимного класса      [lesson\\_ch10/ch/ex12a/ex14/access](#)
  - в качестве конструктора задается имя интерфейса
- ```
public interface IMonster {
    void menace();
    void destroy();
}
void kill();
}
public interface IVampire extends IMonster {
    void drinkBlood();
}
IVampire vent = new IVampire() {
    @Override
    public void drinkBlood() {
    }
    @Override
    public void destroy() {
    }
    @Override
    public void kill() {
    }
    @Override
    public void menace() {
    }
};
```

## Анонимный внутренний класс в методе доступ внешним переменным

- Анонимный внутренний клас может использовать только final переменные
  - чтобы обойти это ограничение и использовать переменные напрямую
  - создается локальная final переменная «прокладка», которая и используется в анонимном классе
- Пример. реализация использования переменной внешней в анонимном классе

```
interface IShow {
    String showInteger();
}

public static void main(String[] args) {
    for (int i = 0; i < 100; i++) {
        final int progress = i;
        System.out.println(new IShow() {

            @Override
            public String showInteger() {
                return progress+"";
            }
        }.showInteger());
    }
}
```
- **ВНИМАНИЕ.** Для доступа к переменным из анонимного класса ИСПОЛЬЗОВАТЬ final прокладки

## Паттерн Фабричный метод

- Применение паттерна с анонимными классами
  - преобразование интерфейсов в анонимные классы
  - улучшение за счет встраивания анонимных классов на базе IEquipment в класс IService
  - Пример
- ```
public interface IService {
    void method1();
    void method2();
}

public interface IFactory {
    IService getService();
}

public class Equipment1 implements IService { // service for Fabric
    @Override
    public void method1() {
        System.out.println("Equipment1.method1");
    }
    @Override
    public void method2() {
        System.out.println("Equipment1.method2");
    }
    public static IFactory factory = new IFactory() { // генератор объектов IService
        public IService getService() {
            return new Equipment1();
        }
    };
    public static IService getService() { // статический метод
        return new Equipment1();
    }
}

public static void main(String[] args) {
    System.out.println("\nFactory Pattern IService Anonymous Class Check\n");
    IService eq1 = Equipment1.factory.getService(); // работа с объектом IFactory
    IService eq2 = Equipment2.factory.getService(); // аналогичный класс
}
```

## Вложенные классы

- Статический внутренний класс называется вложенный класс [lesson\\_ch10/ch/ex16/ex18/ex19](#)
  - для создания объекта не нужен объект внешнего класса
  - **ВНИМАНИЕ.** Интерфейс единственный способ вытаскивать private внутренние классы во вне
  -
- Отличие от внутреннего класса
  - внутренний класс объявляется обычным
  - внутренний класс имеет доступ к полям и методам внешнего класса
  - ??? внутренний класс имеет область видимости только в пределах внешнего класса ???
  - вложенный класс объявляется static
  - вложенный класс не имеет доступа к полям и методам внешнего класса если он не static
  - ??? вложенный класс имеет область видимости не ограниченную внешним классом ???
- Пример. Тройное вложение вложенные классы [lesson\\_ch10/ch/ex20](#)
  - **ВНИМАНИЕ.** все закрытые вложенные классы достаются через интерфейсы
  -

```
public interface IDestination {  
    String getLabel();  
    IEntry getEntry();  
}  
public interface IEntry {  
    void g();  
}  
public class PostOffice {  
    private static class ParcelDest implements IDestination { // static вложенный класс  
        private String label;  
        public ParcelDest(String label) {  
            this.label = label;  
        }  
        @Override  
        public String getLabel() {  
            return label;  
        }  
        private static class EntryLevel implements IEntry{ // static вложенный класс  
            private static int valueY = 25;  
            public void g(){  
                System.out.println("Func.g.ValueY." + valueY);  
            }  
        } // static EntryLevel  
        @Override  
        public IEntry getEntry() { // создать экземпляр  
            return new EntryLevel();  
        }  
    } // static ParcelDest  
  
    public static IDestination getClassDest(String s) {  
        return new ParcelDest(s);  
    }  
    public static IEntry getClassEntry() {  
        return (new ParcelDest("")).getEntry();  
    }  
}  
public static void main(String[] args) {  
    System.out.println("\nExercise 20 Nested Class Check\n");  
    IDestination d = PostOffice.getClassDest("YellowStone"); //экземпляр вложенного класса  
    System.out.println(d.getEntry()); //экземпляр вложенного класса  
    PostOffice.getClassEntry().g(); //экземпляр вложенного класса
```

- Пример. Тройное вложение внутренние классы

[lesson\\_ch10/ch/ex20](#)

- ВНИМАНИЕ. все закрытые внутренние классы достаются через интерфейсы

о

```

public interface IDestination {
    String getLabel();
    IEntry getEntry();
}

public interface IEntry {
    void g();
}

public class MailOffice {
    private class MailDest implements IDestination { // внутренний класс
        private String label;
        public MailDest(String label) {
            this.label = label;
        }
        @Override
        public String getLabel() {
            return label;
        }
        private class HighLevel implements IEntry{ // внутренний класс
            private int valueY = 75;
            public void g(){
                System.out.println("Func.g.ValueY." +valueY);
            }
        } // внутренний класс
        @Override
        public IEntry getEntry() { // создать экземпляр
            return new HighLevel();
        }
    } // внутренний класс MailDest
    public IDestination getClassDest(String s) {
        return new MailDest(s);
    }
}
public static void main(String[] args) {
    System.out.println("\nExercise 20 Internal Class Check\n");
    IDestination md = new MailOffice().getClassDest("OrangeApple"); // внутренний класс
    System.out.println(md.getEntry()); // внутренний класс
}

```

## Классы внутри интерфейсов

- Классы внутри интерфейсов всегда вложенные классы
  - класс внутри интерфейса по умолчанию `public static` и значит он вложенный класс
  - доступ к нему и к его методам также как к обычному классу но с адресом `Interface.ClassName`
  - доступ к `static` методам интерфейса как в обычном классе, но с адресом `Interface.methodName`
- Пример. Интерфейс и вложенный в него классы
- Тестирование общего интерфейса и вложенного класса Test

[lesson\\_ch10/ch/ex21](#)

## Статический класс Main() внутри интерфейса

- Организация `main()` программы внутри интерфейса
  - вложенные классы допускают статические методы
  - поэтому можно добавить в интерфейс свой класс с `main()`
  - **ВНИМАНИЕ.** в IDEA не работает по умолчанию, но запускается
  -
- ```
public interface IClassInterface { // основной интерфейс с вложенным классом внутри
    void howdo();
    static IClassInterface getTest(String s) { // static метод экземпляр класса Test
        return new Test(s);
    }
    static ITest getTest() { // static метод экземпляр класса Test2
        return new Test2();
    }
    class Main {
        public static void main(String[] args) {
            System.out.println("Test Main inside Interface");
        }
    }
}
```

## Доступ наружу из многоократно вложенного класса

- Доступ к методам из вложенного класс
  - осуществляется автоматом по всей вертикали
- Доступ к закрытым вложенным классам только через интерфейсы
  - для генерации объектов использовать интерфейс наружу и последовательные операторы `new`
- Пример .  

```
static ICart getCartStatic() {
    return new Gun().new Magazine().new Cartridge();
}
```

[lesson\\_ch10/ch/ex23/local](#)

## Внутренние классы Назначение

- Внутренний класс способен наследовать независимую реализацию интерфейса
  - Пример. реализации интерфейса в двух вложенных классах [lesson\\_ch10/ch/ex23/include](#)
- ```
public interface ICard {  
    void status();  
}  
  
public interface ICardHolder {  
    static ICard getCardVTB() {  
        return new CardHolder().new CardVTB(); // выдача закрытого вложенного класса  
    }  
    static ICard getCardSB() {  
        return new CardHolder().new CardSB(); // выдача закрытого вложенного класса  
    }  
}  
  
class CardHolder {  
    private class CardVTB implements ICard { // закрытый класс  
        public void status() { // реализация вариант 1  
        }  
    }  
    private class CardSB implements ICard { // закрытый класс  
        public void status() { // реализация вариант 2  
        }  
}
```

## Реализация двух интерфейсов в 3 реализациях

- два интерфейса A и B реализованы в трех классах X, Y, Z [lesson\\_ch10/ch/ex23/remote](#)
  - класс X стандарт, класс Y стандарт и анонимный класс, класс Z оба через анонимные классы
  - ```
public interface InterfaceA {  
    void initA();  
    void playA();  
}  
  
public class ClassX implements InterfaceA, InterfaceB {  
    public void initA() {}  
    public void initB() {}  
    public void playB() {}  
    public void playA() {}  
}  
  
public class ClassY implements InterfaceA { // комбинированная реализация  
    public void initA() {} // стандартная реализация interface A  
    public void playA() {}  
    public InterfaceB getB() { // анонимный класс interface B  
        return new InterfaceB() {  
            public void initB() {}  
            public void playB() {}  
        };  
    }  
    public class ClassZ { // интерфейсы A,B  
        public InterfaceA getA() { // анонимный класс interface A  
            return new InterfaceA() {  
                public void initA() {}  
                public void playA() {}  
            };  
        }  
        public InterfaceB getB() { // анонимный класс interface B  
            return new InterfaceB() {  
                public void initB() {}  
                public void playB() {}  
            };  
        }  
    }  
}
```
  - Собственно вызов интерфейсов
- ```
System.out.println("\nInner Class Multiple Realization Check\n");  
Play.playA(new ClassX()); // прокатило A  
Play.playB(new ClassX()); // прокатило B  
Play.playA(new ClassY()); // прокатило A  
Play.playB(new ClassY().getB()); // прокатило B через внутренний класс  
Play.playA(new ClassZ().getA()); // прокатило A  
Play.playB(new ClassZ().getB()); // прокатило B через внутренний класс
```

## Реализация множественного наследования с простыми и абстрактными классами

- Реализация интерфейсов при наличии абстрактного и обычного класса [lesson\\_ch10/ch/ex23/integra](#)
  - простой класс наследуется стандартным способом `extends`
  - абстрактный класс наследуется через анонимный внутренний класс
- Реализация интерфейса множественное наследование простых и абстрактных классов
  - абстрактные классы сразу переводятся в анонимные внутренние классы
  - простые классы наследуются абстрактными классами и затем переводятся в анонимные классы
  - анонимные классы размещаются внутри класса обеспечивая множественное наследование
  - Пример.

```
public class ClassD {  
    public void show() {  
        // обычный класс  
        System.out.println("ClassD.show");  
    }  
}  
  
abstract public class ClassE {  
    public abstract void show();  
}  
  
public class ClassF extends ClassD { // обычный класс D наследуется extends  
    public ClassE getE() { // для работы по интерфейсу D  
        return new ClassE(); // абстрактный класс наследуется анонимным классом  
        public void show() { // для работы по интерфейсу E  
            System.out.println("ClassF.show");  
        };  
    }  
}  
  
abstract public class ClassG extends ClassD{ // абстрактный класс прокладка  
    // для наследования D  
}  
  
• public class ClassH {  
    public ClassE getE() { // абстрактный класс наследуется анонимным классом E  
        return new ClassE() {  
            @Override  
            public void show() {  
                System.out.println("ClassE.ClassH.show");  
            };  
        }  
    }  
    public ClassG getD() { // абстрактный класс наследуется анонимным классом D  
        return new ClassG() {  
            @Override  
            public void show() {  
                System.out.println("ClassD.ClassH.show");  
            };  
        };  
    }  
}  
  
public class Play {  
    public static void playD(ClassD d) {  
        d.show();  
    }  
    public static void playE(ClassE e) {  
        e.show();  
    }  
}  
  
• public static void main(String[] args) {  
    System.out.println("\nInner Class Multiple Realization Check\n");  
    Play.playD(new ClassF()); // прокатило  
    Play.playE(new ClassF().getE()); // прокатило  
    Play.playD(new ClassH().getD()); // прокатило  
    Play.playE(new ClassH().getE()); // прокатило
```

## Свойства внутренних классов

- Внутренний класс может иметь любое число экземпляров, внутреннее содержание каждого не зависит от состояния внешнего объекта
- Внешний класс может иметь сколько угодно внутренних классов, каждый со своей реализацией интерфейса, либо все наследуют один интерфейс
- Место создания объекта внутреннего класса не привязано к созданию внешнего объекта
- Внутренний класс не работает как потомок внешнего класса, это независимая сущность
  - связь внутреннего класса с внешним происходит при создании объекта внутреннего класса
  - ВНИМАНИЕ. Доступ из внутреннего класса во внешний по ссылке ExtClassName.This
- Пример.

[lesson\\_ch10/ch/ex23/access](#)

```
public interface ISelector {  
    boolean checkEnd();  
    Object getCurrent();  
    void nextIndex();  
    static void play(ISelector s) { // работа с объектом Selector внутреннего класса  
        System.out.print(s.getCurrent() + " ");  
        s.nextIndex();  
    }  
}  
  
public static void main(String[] args) {  
    System.out.println("\nExercise 23 Inner Class Check\n");  
    Sequence sq = new Sequence(20);  
    ISelector sf = sq.getForward(); // здесь подключаются внутренний и внешний классы  
    ISelector sr = sq.getReverse();  
    ISelector.play(sf); // через static  
    sq.play(sr); // работа внешнего класса с внутренним и внутреннего с внешним
```

## Замыкания и обратные вызовы

### Замыкание

[lesson\\_ch10/ch/ex25/local](#)

- Замыкание это вызываемый объект который сохраняет информацию о контексте где был создан
  - например внутренний класс, имеет доступ к членам, методам и ссылку на объект внешнего класса
  - в общем когда получаем экземпляр внутреннего класса это ВСЕГДА ЗАМЫКАНИЕ
- Пример. Closure это замыкание, он имеет доступ к ссылке внешнего объекта CallTwo.this

```
private class Closure implements Incrementable { // non static объект  
    public void increment() {  
        CallTwo.this.increment(); // вызываем переопределенную CallTwo  
    }  
    public Incrementable getCallBack() { // создать внутренний класс как объект  
        return new Closure(); // closure замыкание на объект внешнего класса  
    }  
    public Incrementable getI() { // анонимный класс из интерфейса напрямую  
        return new Incrementable() { // closure пример получаем замыкание  
            @Override  
            public void increment() {  
                CallTwo.this.increment(); // closure доступ к объекту внешнего класса  
                CallTwo.super.increment();  
            };  
        };  
    };
```

## Замыкание и static context

- Создание объекта из static контекста, ЭТО УЖЕ НЕ ЗАМЫКАНИЕ

lesson\_ch10/ch/ex24/local

```
private static class Closure2 implements Incrementable { // static context
    @Override
    public void increment() {
        System.out.print("CallTwo.Closure.increment > ");
        CallTwo.this.increment(); // не работает в static context поэтому это НЕ замыкание
    }
}

public static Incrementable getCallBack2() { // создаем объект внутреннего класса
    return new Closure2();
}

public static Incrementable getIStatic() { // это просто внутренний класс
    return new Incrementable() {
        @Override
        public void increment() {
            System.out.print("CallTwo.Incrementble.increment static");
        }
    };
}
```

## Обратный вызовCallBack

- CallBack это вызов метода текущего объекта посторонним объектом, созданным текущим объектом
  - Пример. CallTwo создает объект new Incrementable(){}, далее new Incrementable(){} вызывает
  - метод CallTwo.this.increment() объекта CallTwo, создавшего объект new Incrementable(){}
    - прямой вызов CallTwo.increment()
    - callBack вызов CallTwo >> new Incrementable(){} >> { CallTwo.this. increment } >> CallTwo.increment()

## Внутренние классы и control framework

- application framework
  - это набор классов для решения определенного круга задача
  - для адаптации наследуется один или несколько классов и переопределяются методы
- control framework
  - это один из application framework для построения event driven system системы отработки событий
  - как пример event driven system есть система GUI графический интерфейс пользователя
- event driven system EDS
  - определить интерфейс работы с событиями системы
  - создать обработчик событий
  - использовать абстрактные классы в интерфейсе и обработчике для универсальности реализации
- реализация EDS
  - система оранжереи,
  - события это включение света, воды, нагревателей, сигнализация, перезапуск системы
  - программа пользователя ставит события в очередь
  - контроллер событий ставит в очередь автозагружаемые события
  - контроллер событий проверяет очередь, выполняет событие, удаляет событие
  - **ВНИМАНИЕ.** Выход из программы жесткий по System.exit(0)
- реализация EDS расширенная версия
  - добавлен потомок контроллера с новыми классами событий
  - добавлена система мягкого выхода из программы по флагу, а не системным выходом
- 

## Наследование от внутренних классов

- Наследование внутренних классов сложнее
  - потому что конструктор внутреннего класса должен подключить объект внешнего класса
  - то есть надо сначала создать объект внешнего класса, а потом наследовать внутренний класс
- Синтаксис состоит из имени класса наследования и формата конструктора
  - имя класса наследования class NewClass extends ExtClass.InnerClass { }
  - формат конструктора NewClass { new ExtClass( param).super( param); }
  - Пример. конструкторы с параметрами
  -
- ```
public class HomeRoad extends Fitness.RunnerRoad {
    private String type;
    private int noise;
    public HomeRoad(String fName, String rName, int weight, String hRoad, int noise) {
        new Fitness(fName).super(rName, weight);
        this.type = hRoad;
        this.noise = noise;
        System.out.println("HomeRoad.Constructor");
    }
}
```

## Переопределение внутреннего класса НЕ РАБОТАЕТ

- Переопределение внутреннего класса не происходит
  - **ВНИМАНИЕ.** как и с полями, наследование не работает с внутренними классами
  - При наследовании появляется два внутренних класса доступные через this и super

## Наследование парное внешних и внутренних классов

- При парном наследовании
  - работает переопределение методов
  - работает переопределение конструкторов в каком контексте создать объект тот и конструктор
  - Пример. парное наследование Egg.Yolk => BigEgg.Yolk
    - конструктор Egg.Yolk срабатывает если new Yolk идет в контексте Egg
    - конструктор BiggEgg.Yolk срабатывает если newYolk запускается в контексте BigEgg.

[lesson\\_ch10/ch/ex28/include](#)

## Локальные внутренние классы

- локальные внутренние классы могут создаваться в блоках кода или метода
- локальный класс имеет доступ к final текущего блока, метода, а также ко всем полям внешнего класса
- **ВНИМАНИЕ.** Локальные классы применяют вместо анонимных, когда нужно несколько экземпляров
- Пример

```
public class Intern {  
    private int value = 15;  
    private String s = "Track";  
    void init() {  
        final int mvalue = 23;  
        final String sm = "Repo";  
        class InnerClass {  
            void demo() {  
                System.out.println(value++ + " "+s+" "+mvalue + " "+sm);  
                System.out.println(value++ + " "+s+" "+mvalue + " "+sm);  
            }  
            InnerClass ir = new InnerClass();  
            ir.demo();  
        }  
    }  
}
```

- Пример. Реализация интерфейса

- **ВНИМАНИЕ.** Элегантное решение через анонимный и локальный классы [lesson\\_ch10/ch/ex28/remote](#)

```
public class LocalClass {  
    private int count = 0; // закрытая переменная класса  
    ICounter getCountInner(String nameCounter) { // интерфейс через локальный класс  
        class InnerClass implements ICounter {  
            public InnerClass() {  
                System.out.println("InnerClass.Constructor");  
            }  
            @Override  
            public int next() {  
                System.out.println(nameCounter + ". " + ++count); // распечатать значение  
                return count;  
            } // локальный класс  
            return new InnerClass();  
        } // метод  
    } // метод  
    ICounter getCountAnonim(String nameCounter) { // интерфейс через анонимный класс  
        return new ICounter() {  
            @Override  
            public int next() {  
                System.out.println(nameCounter + ". " + ++count); // распечатать значение  
                return count;  
            }; // анонимный класс  
    } // метод
```

## Идентификаторы внутренних классов

- Внутренние классы имеют идентификатор \$ при компиляции
  - внутренние классы чередуются значком \$ и именем класса
  - анонимные классы чередуются значком \$ и порядковым номером класса
  - файлы классов вышеуказанного проекта LocalClass\$1InnerClass.class, LocalClass\$1.class

## Урок 11

### Коллекции

- Коллекция это класс организованный как массив, который может содержать объекты любого типа
  - доступ к элементу в коллекции определяется методами класса коллекции
  - для безопасности применяются обобщенные, которые ограничивают тип объектов в коллекции

### Обобщенные типы Generics

- Обобщенные типы или Generic используются для ограничения типов в коллекции [lesson\\_ch11/ch/ex1](#)
  - класс с обобщенным типом принимает какой то тип при создании объекта и
  - после чего объект может работать только с данными указанного типа
  - это предотвращает проникновение в коллекцию объектов других типов и избежать исключений
  - Пример
- ```
void checkCollection() {
    ArrayList<Apple> apples = new ArrayList();
    apples.add(new Apple()); // добавить три яблока
    apples.add(new Apple());
    // apples.add(new Orange()); // на этапе компиляции не пропустит
    for (Object apple : apples) {
        System.out.println((Apple)apple.getId());
    }
}
```
- Generics и Наследование [lesson\\_ch11/ch/ex1/include/access](#)
  - Generics типы отлично работают с восходящим преобразованием, т.е. с потомком в теле предка
  -

### Основные концепции

- Collection есть две разновидности List и Set [lesson\\_ch11/ch/ex2/ex3/access](#)
- List список в определенной последовательности
  - ArrayList<>() элементы в порядке вставки, как массив
  - LinkedList<>() элементы в порядке вставки, как список, поддерживает больше операций
  - **ВНИМАНИЕ.** LinkedList не совсем совместим с предком List т.к. содержит новые методы
- Set множество, тоже список, но можно добавлять по одному элементу, не повторяющиеся элементы
  - HashSet<>(); самый быстрый доступ
  - TreeSet<>(); хранит элементы по возрастанию
  - LinkedHashSet(); хранит элементы в порядке добавления
- Queue очередь список с доступом который называется дисциплина очереди
  - элементы вставляются в список с одной стороны и извлекаются с другого конца
- Map список где каждый элемент хранит два значения ключ и объект
  - это массив значений, доступ к которым идет по массиву ключей
  - HashMap(); самый быстрый доступ
  - TreeMap(); ключи отсортированы по возрастанию
  - LinkedHashMap(); ключи хранятся в порядке вставки
  - **ВНИМАНИЕ.** TreeMap не совсем совместим с предком Map т.к. содержит новые методы
- Для перебора элементов используется механизм Iterator
  - foreach{} запускает механизм Iterator, вызывается через <iter>

## Добавление групп элементов

- Collection и Arrays есть методы добавления группы элементов [lesson\\_ch11/ch/ex4/local/include](#)
- Arrays.asList() принимает массив или список элементов и создает Collection или List объект
  - <Collection> использовать в качестве аргумента для ArrayList, Collection.addAll
  - <List> создает объект типа List, но размер фиксированный при создании
- **ВНИМАНИЕ.** Arrays.asList() может некорректно работать с потомками ??? не подтверждается ???
  - Collections.addAll() предпочтительно
- Collections.addAll() добавляет массив или список объектов и добавляет в Collection
  - работает быстро, но не создает объект
  - **ВНИМАНИЕ.** Предпочтительно создать пустой Collection и вызвать Collections.addAll()
- Collection.addAll() требует в качестве аргумента только другой Collection
  - **ВНИМАНИЕ.** collection.addAll() не такой гибкий, поэтому применяется реже
- ```
List<Snow> snow1 = Arrays.asList(      // создаем объект List    все объекты extends Snow
        new Crusty(), new Slush(), new Powder()
);
List<Snow> snow2 = Arrays.asList(      // создаем объект List    все объекты extends Snow
        new Light(), new Heavy(), new Powder(), new Crusty(), new Slush()
);
List<Snow> snow3 = new ArrayList<>();    // создаем объект List    все объекты extends Snow
Collections.addAll(snow3,new Light(), new Heavy(), new Crusty()); // через collection
List<Snow> snow4 = Arrays.<Snow>asList( // создаем объект List    все объекты extends Snow
        new Light(), new Heavy(), new Powder(), new Crusty(), new Slush()
); // через явное указание типа
```
- **ВНИМАНИЕ.** snow4 создан через явное указание аргумента
- **ВНИМАНИЕ.** Методы добавления группы создания списка через Collections или через кастинг в asList()

## Вывод контейнеров

- вывод контейнеров на печать [lesson\\_ch11/ch/ex4/integra](#)
  - Arrays.toString() мощная функция на печать коллекций, только добавить toString() для элемента
  - контейнеры выводятся на печать и самостоятельно, если это <String>
- Пример

```
public class Animals {
    ArrayList<String> alist = new ArrayList<>();
    LinkedList<String> llist = new LinkedList<>();
    HashSet<String> hset = new HashSet<>();
    TreeSet<String> tset = new TreeSet<>();
    LinkedHashSet<String> lhset = new LinkedHashSet<>();
    HashMap<String, String> hmap = new HashMap<>();
    LinkedHashMap<String, String> lhmap = new LinkedHashMap<>();
    TreeMap<String, String> tmap = new TreeMap<>();
    public Collection fill(Collection<String> collect) {
        collect.add("rat");
        collect.add("cat");
        collect.add("dog");
        collect.add("mouse");
        collect.add("dog");
        return collect;
    }
    Map fill(Map<String, String> map) {
        map.put("rat", "Anni");
        map.put("cat", "Mangi");
        map.put("dog", "Greg");
        map.put("mouse", "Lori");
        map.put("dog", "Bob");
        return map;
    }
}
```

## Итератор

- Итератор, это объект доступа по индексу к контейнеру Collection
  - итератор вызывается с помощью метода `Collection.iterator()`, проходит список от начала до конца
  - **ВНИМАНИЕ.** Для сброса итератора надо ПРОСТО СОЗДАТЬ ЕГО ЗАНОВО

## List

- Контейнер List имеет две разновидности `ArrayList` и `LinkedList` [lesson\\_ch11/ch/ex5/](#)
  - `ArrayList` быстро произвольный доступ, медленно вставка, удаление в середине списка
  - `LinkedList` медленно произвольный доступ, быстро вставка удаление в середине списка
- `ArrayList` методы [lesson\\_ch11/ch/ex6/access](#)
  - `contains()` проверяет, есть ли объект в списке
  - `remove()` удалить объект из списка
  - `indexOf()` получить индекс объекта в списке
  - `sublist()` получение части списка
  - **ВНИМАНИЕ.** все изменения `sublist` автоматом отражаются в головном списке, это его живая часть
  - `sort()` сортировка списка
  - `shuffle()` перемешивание списка
  - `containsAll()` проверяет содержится ли данный `sublist` в `list`
  - `retainAll()` оставляет только те элементы которые есть в `sublist`
  - `removeAll()` удаляет все элементы `sublist` из `list`
  - `toArray()` клонирует элементы списка в массив, если надо расширяет
  - **ВНИМАНИЕ.** если массив меньше, то расширяется, если больше, заполняется, остальное null

- `sublist()`

```
List<Integer> sub = ints.subList(1, 4);      // sublist [1,2,3] 4 не входит
System.out.println(" : ints>" + ints);        // это живая часть списка головного списка
sub.add(121);                                // добавляет в головной список 121
System.out.println(" : add>" + ints);
sub.set(1, 543);                             // меняет [1] на 543 в головном списке
System.out.println(" : set>" + ints);          // головной список
System.out.println(" : sub>" + sub);           // частичный список
// проверяет sublist живой для list или нет
System.out.println("10: containsAll()>" + ints.containsAll(sub));
```

- `toArray()`

```
Object[] o = pets.toArray();
System.out.println("22: toArray()>" + Arrays.toString(o)); // создали массив
String[] pa = pets.toArray(new String[1]); // расширение массива
System.out.println("23: toArray(new [1])>" + Arrays.toString(pa) + " id>" + pa[0]);
String[] pb = pets.toArray(new String[8]); // автозаполнение массива
System.out.println("24: toArray(new[8])>" + Arrays.toString(pb) + " id>" + pb[1]);
```

- `asList()`

**ВНИМАНИЕ.** всегда создает список из массива объектов, не примитивов, а именно объектов

```
List<Integer> ints = arrayList(7);           // создание при помощи утилиты инициализации
```

```
Integer[] aInts = ArrayUtils.toObject(Range.rangeInt(7)); // конвертер int[] >> Integer[]
List<Integer> ints = new ArrayList(Arrays.asList(aInts)); // список из массива объектов
```

## Работа со списком

- Создать список из массива, выделить sublist и удалить его
  - класс хранения Ball , каждый объект имеет цвет выбран случайно при создании в конструкторе
  - массив Ball заполняется своим статическим методом,на входе пустой массив, на выходе заполнен
  - объект Ball, вывод на печать метод toString

```
public class Ball {  
    private final String[] S_COLOR = new String[]{  
        "brown",  
        "black",  
        "red",  
        "blue",  
        "yellow",  
        "green",  
        "magenta",  
        "purple",  
        "orange",  
        "white"  
    };  
    private Random rnd = new Random();  
    String color;  
  
    public static Ball[] getBalls(Ball[] balls) {  
        for (int i = 0; i < balls.length; i++) {  
            balls[i] = new Ball();  
        }  
        return balls;  
    }  
    public Ball() {  
        this.color = S_COLOR[rnd.nextInt(10)];  
    }  
  
    @Override  
    public String toString() {  
        return "Ball." + color;  
    }  
}  
public static void app() {  
    System.out.println("\n=====ACCESS====");  
    System.out.println("\nExercise Check\n");  
    Ball[] balls = getBalls(new Ball[10]); // создать и заполнить массив на 10 объектов  
    List<Ball> list = new ArrayList<Ball>(Arrays.asList(balls));  
    System.out.println("Ball array.done >" + Arrays.toString(balls)); // массив создали  
    System.out.println("Ball list.done >" + list); // список создали  
    List<Ball> sub = list.subList(1, 5); // sublist создали  
    System.out.println("Ball sublist >" + sub);  
    list.removeAll(sub); // sublist из списка удалили  
    System.out.println("Ball remove() >" + list);  
}
```

lesson\_ch11/ch/ex7/access

## Итераторы

- Iterator это к тому же паттерн проектирования
- Итератор это объект который обеспечивает перемещение по спискам
  - списки могут быть любого типа, для всех можно сделать доступ через интерфейс <Iterator>
  - итератор занимает мало ресурсов
- Iterator методы
  - iterator()      собственно запрос итератора
  - next()            получить следующий элемент
  - hasNext()        запрос на наличие следующего элемента
  - remove()        удалить элемент по последней next()
- Пример. работа с Iterator

```
public static void app() {  
    List<Pet> pets = Pets.arrayList(12); // не использовать new!!!  
    Iterator<Pet> it = pets.iterator(); // получили итератор обязательно <Pet>  
    System.out.print("Iterator manual > ");  
    while (it.hasNext()) {  
        Pet p = it.next(); // обязательно <Pet> в итератор чтобы здесь сработало  
        System.out.print(p.id()+":"+p+" ");  
    }  
    System.out.print("\nIterator foreach > ");  
    for (Pet pet : pets) { // тоже самое с итератором, но неявно  
        System.out.print(pet.id()+":"+pet+" ");  
    }  
    it = pets.iterator(); // пересоздать итератор это нормально, они одноразовые  
    for (int i = 0; i < 6; i++) {  
        it.next(); // получить объект  
        it.remove(); // удалить  
    }  
    System.out.print("\nIterator removal > ");  
    for (Pet pet : pets) { // тоже самое с итератором, но неявно  
        System.out.print(pet.id()+":"+pet+" ");  
    }  
    System.out.println();  
}
```

## Итераторы как универсальный доступ к содержимому

- Итераторы позволяют получать доступ к контейнерам абсолютно разных типов [lesson\\_ch11/ch/ex11](#)
  - в функцию обработчика отдается только итератор
- Пример. разные коллекции используют один и тот же метод

```
public class Collect {  
    ArrayList<String> aList = new ArrayList<>(Arrays.asList( "Heavy", "Link", "Word"));  
    LinkedList<Integer> lList = new LinkedList<>(Arrays.asList( 120, 21, 75, 93, 32, 567));  
    HashSet<Rodent> hSet = new HashSet<>(Arrays.asList(getRodentArray(new Rodent[8])));  
    TreeSet<Ball> tSet = new TreeSet<>(Arrays.asList(Ball.getBalls(new Ball[12])));  
  
    void show(Iterator it) {  
        while (it.hasNext()) {  
            System.out.print(it.next()+" ");  
        }  
        System.out.println();  
    }  
}  
• public class Access {  
    public static void app() {  
        Collect c = new Collect();  
        c.show(c.aList.iterator());  
        c.show(c.lList.iterator());  
        c.show(c.hSet.iterator());  
        c.show(c.tSet.iterator());  
    }  
}
```

[lesson\\_ch11/ch/ex8/ex9/ex10](#)

## Iterator удаление элементов во время перебора

- Iterator позволяет удалять элементы во время перебора
  - используется метод Iterator.remove() который удаляет текущий элемент итератора
- **ВНИМАНИЕ.** Iterator ДОПУСКАЕТ изменения в списке внутри цикла.
- ForEach НЕ ПОЗВОЛЯЕТ удалять элементы во время перебора
  - пользователь не имеет доступа к итератору,
  - удаление производится напрямую из списка List.remove(object), что вызывает Exception
- **ВНИМАНИЕ.** ForEach НЕ ДОПУСКАЕТ изменения в списке внутри цикла, только ДО или ПОСЛЕ цикла.
- Пример. реализация удаления элементов в Iterator и ForEach
- ```
public static void main(String[] args) {  
    List<String> list = new ArrayList<>();  
    String s = "abcdefghijklmnopqrstuvwxyz";  
    for (int i = 0; i < s.length(); i++) {  
        list.add(s.substring(i, i + 1));  
    }  
  
    List<String> list2 = new ArrayList<>(list);  
  
    Iterator<String> it = list.iterator();  
    int count = 0;  
    while (it.hasNext()) { // так работает  
        String sNext = it.next();  
        System.out.println(sNext);  
        if (count++ % 3 == 0) {  
            it.remove();  
        }  
    }  
    System.out.println(list);  
  
    count = 0;  
    for (String s2 : list2) { // так не работает  
        System.out.println(s2);  
        if (count++ % 3 == 0) {  
            list2.remove(s2);  
        }  
    }  
}
```

## ListIterator

- ListIterator это интерфейс к итераторам для списков
  - допускает двухсторонние движения по спискам
  - **ВНИМАНИЕ.** для установки итератора в конец списка надо СОЗДАТЬ его с параметром list.size()
  - у него есть дополнительно методы prev(), set()
  - prev() предыдущий элемент
  - set() заменить текущий элемент
- Проход от конца к началу списка
  - создать ListIterator с индексом равным длине списка
  - идти от конца к началу с методом it.previous()
- Пример

```
List<Integer> list = Range.arrayList(10);
List<Integer> list2 = new ArrayList<Integer>(list); // сделать копию
ListIterator <Integer> it = list.listIterator();
ListIterator <Integer> it2 = list2.listIterator(list2.size()); // ставим в конец списка
System.out.println(list);
System.out.println(list2);
while (it.hasNext()) { // it.next() идет от начала к концу
    it2.previous(); // it2.previous() идет от конца к началу
    it2.set(it.next()); // переписываем list2 ставим в конец списка
}
```

[lesson\\_ch11/ch/ex12/access](#)

## LinkedList

- LinkedList и ArrayList реализуют базовый интерфейс List
  - содержит методы работы как со стеком,
  - быстрее работает с операциями удаления вставки в середине списка
  - медленнее работает с операциями произвольного доступа
- LinkedList методы
- Вернуть элемента списка
  - peek() первый элемент списка
  - getFirst(), element() тоже самое
  - get(int) элемент по индексу
- Удалить и Вернуть элемент списка
  - poll() первый элемент списка
  - removeFirst(), remove() тоже самое
  - removeLast() последний элемент списка
- Добавить элемент списка
  - addFirst() в начало списка первый элемент
  - offer() в конец списка, последний элемент
  - add(), addLast() тоже самое
- Пример. вставка в центр списка

```
public static void app() {
    LinkedList<Integer> list = new LinkedList<>(); // пустой контейнер
    ListIterator<Integer> it = list.listIterator();
    it.add(Range.getInt(100)); // два значения
    it.add(Range.getInt(100));
    for (int i = 0; i < 20; i++) {
        it = list.listIterator(list.size()/2-1); // установить на середину списка
        it.next(); // активировать элемент
        it.add(Range.getInt(100)); // добавить новый элемент
        System.out.println(list);
    }
}
```

[lesson\\_ch11/ch/ex12a/local](#)

если список пустой

null

exception

exception

null

exception

exception

[lesson\\_ch11/ch/ex14/access](#)

## Queue

- Queue очередь методы LimnkedList для Queue
  - element() вернуть первый элемент
  - peek () вернуть первый элемент
  - poll() удалить и вернуть первый элемент
  - remove() удалить и вернуть первый элемент
  - offer() добавить элемент в конец очереди
  -

[lesson\\_ch11/ch/ex12a/local](#)

## Stack Стек

- Стек это контейнер который работает по принципу LIFO
  - LIFO последний вошел, первый вышел
  - LinkedList позволяет реализовать простой Stack
- Stack методы LinkedList
  - getFirst() вернуть первый элемент >> peek()
  - addFirst() добавить первый элемент >> push
  - removeFirst() удалить и вернуть первый элемент >> pop
  - isEmpty() вернуть пуст ли стек или нет
  -

[lesson\\_ch11/ch/ex15/local](#)

```
public class SimpleStack<T> {    // bay работаем с Generics
    private LinkedList<T> stack = new LinkedList<T>(); // список с определяемым типом
    public void push(T value) {
        stack.addFirst(value);    // добавить в начало
    }
    public T pop() {
        return stack.removeFirst(); // удалить и вернуть из начала
    }
    public T peek() {
        return stack.getFirst(); // вернуть элемент из начала
    }
    public boolean isEmpty() {
        return stack.isEmpty();
    }
    @Override
    public String toString() {
        return stack.toString(); // в строку по умолчанию
    }
}
public class Local {
    public static void app() {
        SimpleStack<Ball> stack = new SimpleStack();
        stack.push(new Ball());
        stack.push(new Ball());
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }
}
```

## Set Множество

- Set множество это контейнер который хранит только уникальные значения
  - используется чаще для проверки на принадлежность значения к множеству
  - не имеет уникальных методов, есть только методы которые есть у Collection
- Методы

[lesson\\_ch11/ch/ex16/local](#)

- add() добавить элемент
- remove(),removeAll() удалить элемент, удалить группу
- contains(),containsAll() есть ли элемент в множестве, есть ли подмножество
- Collections.addAll() добавить группу элементов

```
public static void app() {  
    Set<Integer> set = new LinkedHashSet<>(); // методы быстрого доступа к множеству  
    for (int i = 0; i < 10000; i++) {  
        set.add(Range.getInt(30)); // заполнить 10К случайных значений от 0 до 99  
    }  
    System.out.println(set);  
    Set<String> set1 = new HashSet<>(); // создать объект Set  
    Collections.addAll(set1, new String("А В С Д Е Ф Г Н И Ј К Л").split(" "));  
    set1.add("М"); // добавляем новый элемент  
    System.out.println("set1: contains(\"Н\") > "+ set1.contains("Н")); // новая коллекция  
    Set<String> set2 = new HashSet<>();  
    Collections.addAll(set2, new String("Н И Ј К Л").split(" ")); // добавить символы  
    System.out.println("set1:containsAll(set2) > "+ set1.containsAll(set2)); // есть ли  
    set1.remove("Н"); // удалить объект  
    set1.removeAll(set2);  
    Collections.addAll(set1, new String("Х У З").split(" ")); // добавить символы  
}
```

## Set виды множеств и сортировка

- HashSet неупорядоченный набор значений [lesson\\_ch11/ch/ex22/access](#)
- TreeSet упорядоченный по возрастанию множество значений [lesson\\_ch11/ch/ex16/include](#)
- LinkedHashSet упорядоченное множество значений в порядке добавления
- Методы сортировки через итератор
  - it.hasNext() есть ли элементы в множестве
  - it.next() следующий элемент множества
  - it.remove() удалить текущий элемент множества

### ВНИМАНИЕ. Iterator.next() дает ссылку на объект ВНУТРИ множества

```
HashSet<Word> hset = new HashSet<>(); // работа с быстрым списком  
Iterator<Word> it;  
for (String s : textFile) { // класс TextFile содержит итератор, перебираем  
    Word word = new Word(s);  
    if (hset.contains(word)) { // ищем по строке переписано Equal  
        it = hset.iterator(); // итератор выделяется каждый раз на слово  
        while (it.hasNext()) {  
            Word itWord = it.next(); // ВНИМАНИЕ. получили доступ к объекту  
            if (itWord.equals(word)) {  
                itWord.inc(); // обновить объект  
            }  
        }  
    } else {  
        hset.add(new Word(s));  
    }  
}
```

## **Set и класс хранения пользователя**

- Set множества накладывает ограничение на класс хранения пользователя [lesson\\_ch11/ch/ex16/access](#)
  - HashSet и LinkedHashSet позволяют хранить класс пользователя без ограничений
  - TreeSet требует чтобы класс пользователя наследовал интерфейс Comparable
- **ВНИМАНИЕ.** Если не проверять Comparator, то рассматривает как разные объекты по ссылкам
- даже если значения внутри класса одинаковые
- Может вставить много объектов вида Word("System",1) если это разные ссылки

## **TreeSet Особенности работы**

- Если класс хранения пользователя НЕ содержит Comparator
  - HashSet и LinkedHashSet добавляют новые объекты с теми же значениями если ссылки разные
  - как результат много ДУБЛИРУЮЩИХ объектов
- Если класс хранения пользователя СОДЕРЖИТ Comparator
  - можно применить TreeSet
  - TreeSet НЕ ГАРАНТИРУЕТ корректную работу если Comparator НЕ НАСТРОЕН ВЕРНО
- Пример. Подсчет повторов слов в файле **НЕ РАБОТАЕТ**
  - Class Word, два поля <String> и <Integer>, Comparator = <String>+<Integer>.toString
  - При подсчете ищем объект Word(<\*><1>) и слово "import" добавится как Word(<import><1>)
  - Второй раз, нашли Word(<import><1>), инкремент, Добавили в наш Word(<import><2>)
  - **ВНИМАНИЕ.** В TreeSet будет 2 объекта Word(<import><1>) и Word(<import><2>)
- Пример. Подсчет повторов слов в файле **НЕ РАБОТАЕТ**
  - Class Word, два поля <String> и <Integer>, Comparator = <String>
  - При подсчете ищем объект Word(<\*><1>) и слово "import" добавится как Word(<import><1>)
  - Второй раз, нашли Word(<import><1>), инкремент, Добавили в наш Word(<import><2>)
  - **ВНИМАНИЕ.** В TreeSet не будет добавления или замещения и останется Word(<import><1>)
- Пример. Подсчет количества букв в слове **РАБОТАЕТ**
  - Class Word, два поля <String> и <Integer>, Comparator = <String>
  - Подсчитываем количество букв в данном слове.
  - При подсчете ищем объект Word(<\*><N>) и слово "import" добавится как Word(<import><N>)
  - Второй раз, тоже самое добавляем Word(<import><N>)
  - **ВНИМАНИЕ.** В TreeSet не будет добавления или замещения и останется Word(<import><N>)
  - Поэтому для данного слова все сработает.

## **Comparator работа с классом хранения пользователя**

- Comparator задается интерфейсом Comparable [lesson\\_ch11/ch/ex23/access](#)
  - можно сразу задать тип объектов для функции compareTo()
  -
- ```
public class Number implements Comparable<Number> { // задается сразу тип в compareTo
    private int val;
    private int count;
    @Override
    public int compareTo(Number o) { // сравнение только по значению count
        if(count < o.count ) {
            return -1;
        }
        if(count == o.count ) {
            return 0;
        }
        return 1; // count > remote
    }
}
```

## Map

- Мар это коллекция состоящая из двух полей, ключа и значения
  - позволяют искать объекты по ключам
  - могут быть расширены до нескольких измерений, если в <value> вставить объекты Map
- Методы
  - get()              получить значение по ключу, если нет то null
  - put()              записать значение по ключу
  - keyset()          массив ключей для итератора
  - values()          массив значений для итератора
  - entrySet()        массив пар ключ, значение для итератора
- Пример.            проверка частоты повторов Random()
- число работает как ключ, а значение это число повторов

```
public static void app() {  
    Map<Integer, Integer> map = new HashMap<>(); // создали карту (key, value)  
    for (int i = 0; i < 10000 ; i++) { // ключ значение числа, значение повторы  
        int value = Range.getInt(10); // от 0..19  
        Integer freq = map.get(value); // получить частоту  
        if (freq == null) {  
            freq = 0;  
        }  
        map.put(value, freq+1); // нарастить значение для данного числа  
    }  
    System.out.println(map);  
}
```

[lesson\\_ch11/ch/ex17/local](#)

## Мар несколько измерений

- Мар могут быть расширены до нескольких измерений
  - для этого надо в качестве значений <value> использовать контейнеры List, Set, Map
  - реально вставить в <value> отдельный Мар в качестве элемента и внутри делать поиск по ключу

- Пример

- размещение массива ArrayList в качестве значения
- **ВНИМАНИЕ.** Есть такой же пример с Мар в качестве <value>
- 

- ```
public static void app() {
    System.out.println("\n=====INTEGRA====");
    System.out.println("\nMap MultiDimension Check\n");

    Map<Person, List<? extends Pet>> petPeople = new HashMap<>(); // ВНИМАНИЕ
    двойное применение Generic
    petPeople.put(new Person("Dawn"),
        Arrays.asList(
            new Cymric("Molly"),
            new Mutt("Spot")
        ));
    petPeople.put(new Person("Kate"),
        Arrays.asList(
            new Cat("Shackleton"),
            new Cat("Elsie May"),
            new Dog("Margarett")
        ));
    petPeople.put(new Person("Sofie"),
        Arrays.asList(
            new Pug("Louie"),
            new Cat("Stanford"),
            new Cat("Pinkola")
        ));
    petPeople.put(new Person("Jekky"),
        Arrays.asList(
            new Rat("Freckly")
        ));

    System.out.println("People: keyset() > "+petPeople.keySet());
    System.out.println("People: values() > "+petPeople.values());
    System.out.println("Person Iterators:");

    for (Person person : petPeople.keySet()) {
        System.out.print(person+" has:\t");
    }
    // List<Pet> pets = new ArrayList<>(petPeople.get(person)); // отсюда создать итератор
    // for (Pet pet : pets) { // потом перенести инит часть прямо в итератор

        for (Pet pet : petPeople.get(person)) { // потом перенести инит часть прямо
            System.out.print(" {" +pet+"} ");
        }
        System.out.println();
    }
}
```

[lesson\\_ch11/ch/ex17/integra](#)

## Map Iterator

- Мар не работает напрямую с итераторами, поэтому Мар переводят в Set [lesson\\_ch11/ch/ex17/access](#)
- Мар позволяет использовать разные итераторы
  - keyset().iterator() итератор по ключам
  - values().iterator() итератор по значениям
  - entrySet().iterator() итератор по входам
- **ВНИМАНИЕ.** Set работают с итераторами, Set<Map.Entry<String, Gerbil>> позволяет работать с foreach
- Пример. работы с итератором keyset()
- 

```
public static void app() {
    Map<String, Gerbil> map = new HashMap<>(); // карта
    for (int i = 0; i < 10 ; i++) {
        map.put(Gerbil.getName(8), new Gerbil()); // заполняем
    }
    System.out.println(map);
//    Iterator<String> it = map.keySet().iterator();
//    Iterator<Gerbil> it2 = map.values().iterator();
//    Iterator<Map.Entry<String,Gerbil>> itEntry = map.entrySet().iterator();

    Iterator<String> it = map.keySet().iterator();
    while (it.hasNext()) {
        String key = it.next();
        System.out.println("key:" + key + " value:" + map.get(key));
    }
}
```

- Пример. работы с итератором Entry.Set()

```
public static void app() {
    System.out.println("\n====LOCAL====");
    System.out.println("\nMap Analyze Random Check\n");

    Map<Integer, Integer> map = new HashMap<>(); // создали карту (key, value)
    for (int i = 0; i < 10000 ; i++) { // ключ значение числа, значение повторы
        int value = Range.getInt(10); // от 0..19
        Integer freq = map.get(value); // получить частоту
        if (freq == null) {
            freq = 0;
        }
        map.put(value, freq+1); // нарастить значение для данного числа
    }

    System.out.println(map);
//    Set<Map.Entry<Integer, Integer>> set = map.entrySet(); // стандартный путь
//    for (Map.Entry<Integer, Integer> integerIntegerEntry : set) {
//    }
    for (Map.Entry<Integer, Integer> intEntry : map.entrySet()) { // прямой
        System.out.printf("%4d:%4d %4.1f %%\n", intEntry.getKey(), intEntry.getValue(),
                          (double) Math.abs(intEntry.getValue()-1000)/10);
    }
}
```

## Map сортировка

- Map сортировка в общем случае выполняется в три этапа
  - сначала извлечь ключи в набор Array или Set
  - провести сортировку ключей функциями Arrays.sort() или Collection.sort()
  - создать новую LinkedHashMap и добавить элементы по сортированным ключам из старой Map
- Ключи Map переводятся в массив строк в два этапа, сначалав Set через keyset(), затем массив строк
  - keySet() в массив String[] идет вот так keyset().toStringArray(new String[0])
  - **ВНИМАНИЕ.** Формула не работает без массива нулевой длины (new String[0])
- Map простая автоматическая сортировка
  - делается через TreeMap в один этап
  - создать TreeMap используя в качестве исходных данных Map которую надо отсортировать

## Варианты Map сортировки

- Map сортировка ключей может быть сделана несколькими способами
  - сортировка по массиву строк функцией Arrays.sort(String[])
  - сортировка через TreeMap, сортировка и заполнение идет автоматом
  - сортировка через TreeSet, ключ сортируется автоматом, заполнение вручную
  - сортировка через ArrayList, Collections.sort(), через компаратор автоматом
- ```
public static void app() {
    Map<String, Integer> map = new HashMap<>();
    for (int i = 0; i < 20; i++) {
        map.put(new String(String.format("key_%02d", (i+1))), Range.getInt(100));
    }
//Arrays System.out.println(map); // map не сортированный массив
System.out.println("Array:");
String [] sArray = map.keySet().toArray(new String[0]); // создать String[]
System.out.println(Arrays.asList(sArray)); // на печать массив String[]
Arrays.sort(sArray); // отсортировать строки
Map<String, Integer> lhmap = new LinkedHashMap<>();
for (String s : sArray) {
    lhmap.put(s, map.get(s));
}
//TreeMap
Map<String, Integer> mapSorted = new TreeMap<>(map); // сортировка по ключу
Map<String, Integer> lhmap = new LinkedHashMap<>(mapSorted); //автоматом все
//ArrayList
    ArrayList<Map.Entry<String, Integer>> list = new ArrayList<>(map.entrySet());
    Comparator<Map.Entry<String, Integer>> cmp = //make comparator
        new Comparator<Map.Entry<String, Integer>>() { // создаем свой компаратор
            @Override
            public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2) {
                return o1.getKey().compareTo(o2.getKey());
            }
        };
    list.sort(cmp); // сортировка по компаратору
    LinkedHashMap<String, Integer> lhmapb = new LinkedHashMap<>();
    for (Map.Entry<String, Integer> entry : list) { //заполнение сортированным list
        lhmapb.put(entry.getKey(), entry.getValue());
    }
// TreeSet сортировка
    Set<String> set = new TreeSet<>(map.keySet()); // получить строки сортированные
    System.out.println(set);
    Map<String, Integer> lhmapc = new LinkedHashMap<>();
    for (String s : set) { // заполнение сортированным ключом,
        lhmapc.put(s, map.get(s)); // значение вытаскивается по ключу TreeMap
    }
}
```

## Очередь

- очереди используются для надежного перемещения объектов в памяти и между процессами
- очередь базируется на методах LinkedList
- Очередь это контейнер с порядком записи чтения первым вошел первый вышел FIFO
  - короче записываются данные в начало, а читаются с конца
- Queue методы
|  |  |  |
| --- | --- | --- |
| ○ peek() | выдать первый элемент очереди без удаления | работа с Exception |
| ○ element() | тоже самое | null |
| ○ offer() | записать последний элемент очереди | Exception |
| ○ poll() | выдать и удалить первый элемент очереди | null |
| ○ remove() | тоже самое | Exception |

## Priority Queue

- Priority Queue это очередь где извлекается элемент с наивысшим приоритетом
  - позиция элемента в очереди задается сортировкой при вставке offer() или при извлечении poll()
  - сортировка задается в Comparator.PriorityQueue
- Comparator.PriorityQueue
  - это механизм сортировки элементов, который гарантирует выдачу при вызове peek,poll, remove
  - это класс который реализует интерфейс Comparator<Type> и метод compareTo()
  - вызывается при создании объекта очереди new PriorityQueue( list.size(), new CompClass)
  - **ВНИМАНИЕ.** стандартный класс Collections.reverseOrder()
  - Пример
  -

[lesson\\_ch11/ch/ex28/local](#)

```
public class CompIntR implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
}
○ public class Local {
    public static void app() {
        Random rnd = new Random();
        //PQ wo priority
        PriorityQueue<Integer> pqInt = new PriorityQueue<>();
        for (int i = 0; i < 25; i++) {
            pqInt.offer(rnd.nextInt(100));
        }
        Show.showQ(pqInt);

        //Pq with priority
        List<Integer> ints = Range.arrayList(25);
        pqInt = new PriorityQueue<>(ints);           // default forward
        direction
        Show.showQ(pqInt);

        //Pq with priority Rev
        System.out.print("Custom Reverse:");
        pqInt = new PriorityQueue<>(ints.size(), new CompIntR()); // custom
        Comparator
        pqInt.addAll(ints);
        Show.showQ(pqInt);
        System.out.print("Standard Reverse:"); // with standard comparator
        pqInt = new PriorityQueue<>(ints.size(), Collections.reverseOrder());
        pqInt.addAll(ints);
        Show.showQ(pqInt);
```

## Collection и Iterator

- Iterator позволяет унифицировать доступ к коллекции
  - Iterator унификация идет по классу объекта Iterator<Class>
  - Collections тоже самое, унификация по классу Collection<Class>
  - и тогда неважно какой контейнер используется для хранения объектов класса

[lesson\\_ch11/ch/ex30/local](#)

- Пример

```
public static void showCStr(Collection<String> coll) { // Collection <String> class
    System.out.println("Collection elements:");
    for (String s : coll) {
        System.out.printf("%-14s ", s);
    }
}

public static void showMapSS(Map<String, String> map) { // Map<String, String> class
    System.out.println("Map elements:");
    for (String s : map.keySet()) {
        System.out.printf("%-14s:%-14s ", s, map.get(s));
    }
}

public static void showStr(Iterator<String> it) { // Iterator<String> class
    System.out.println("Iterator elements:");
    while(it.hasNext()) {
        System.out.printf("%-16s", it.next());
    }
}

public static void app() { // application with containers
    ArrayList<String> listA = new TextFile("../lesson_lib/lib/utils/Range.java", "\\\W+");
    ArrayList<String> listB = new TextFile("../lesson_lib/lib/utils>Show.java", "\\\W+");
    TreeSet<String> listKey = new TreeSet<>(new CompStrL());
    listKey.addAll(listA);
    Random rnd = new Random();
    Map<String, String> lhmap = new LinkedHashMap<>();
    for (String s : listKey) {
        lhmap.put(s, listB.get(rnd.nextInt(listB.size()))); //по случайному индексу
    }
    Show.showStr(listKey.iterator()); // TreeSet.iterator()
    Show.showStr(listA.iterator()); // ArrayList.iterator()
    Show.showStr(lhmap.keySet().iterator()); // keyset()<String>.iterator()
    Show.showStr(lhmap.values().iterator()); // values()<String>.iterator()
    Show.showMapSS(lhmap); // Map.keySet().iterator()
    Show.showCStr(listKey); // Collection <String>
    Show.showCStr(lhmap.values()); // Collection <String>
```

## Abstract Collection

- Abstract Collection это создание Collection для своего класса хранения
  - создать свой класс с объектом хранения внутри наследует AbstractCollection
  - реализовать методы iterator(), size()
  - реализовать класс Iterator<Class> методы next(), hasNext(), remove()
  - **ВНИМАНИЕ.** Поддерживает стандартный ForEach Iterator
- Реализация без Abstract Collection, при наследование своего класса пишется свой итератор
  - реализация идет в два этапа
  - создается класс хранения как базовый класс
  - в класс потомок добавляется итератор

[lesson\\_ch11/ch/ex30/include](#)

[lesson\\_ch11/ch/ex30/integra](#)

## Collection

- Реализация интерфейса Collection в своем классе
  - по сути просто реализуются все методы по мере возможности
  - позволяет наследовать свои классы, так как это интерфейс
  - и поддерживает foreach для коллекций
  - **ВНИМАНИЕ.** Поддерживает стандартный ForEach Iterator

[lesson\\_ch11/ch/ex30/access](#)

## Foreach Iterator

- Fforeach работает как с массивами, так и с коллекциями
  - Collection содержит Iterable поэтому поддерживается foreach

[lesson\\_ch11/ch/ex31/local](#)

[lesson\\_ch11/ch/ex31/access](#)

## Iterable

- любой класс реализующий интерфейс Iterable будет поддерживаться ForEach
  - это может быть стандартный класс, например Collection, так и класс пользователя
- Пример. свой класс поддерживающий итераторы

```
public class Rodents {  
    protected Rodent[] rodents = Rodent.getRodent(12); // объекты  
    public Rodents(int size) {  
        this.rodents = Rodent.getRodent(size); // объекты  
    }  
}  
  
public class RodentIt extends Rodents implements Iterable<Rodent>{  
    public RodentIt(int size) { // создает объект нужного формата  
        super(size);  
    }  
    @Override  
    public Iterator<Rodent> iterator() {  
        return new Iterator<Rodent>() {  
            private int index = 0;  
            @Override  
            public boolean hasNext() {  
                return index < rodents.length;  
            }  
            @Override  
            public Rodent next() {  
                return rodents[index++];  
            }  
            @Override  
            public void remove() {  
                throw new UnsupportedOperationException();  
            }  
        };  
    }  
    public static void app() {  
        System.out.println("\nUser Class with Iterable Interface:");  
        RodentIt rodentIt = new RodentIt(24); // 24 грызуна создали  
        for (Rodent rodent : rodentIt) {  
            System.out.printf("%-24s", rodent);  
        }  
    }  
}
```

## Set и Iterable

- Set как часть Collection реализует интерфейс Iterable и поэтому поддерживает ForEach
- Пример

[lesson\\_ch11/ch/ex31/include](#)

```
public static void app() {  
    Set<Map.Entry<String, String>> hset = System.getenv().entrySet(); // набор Set  
    for (Map.Entry<String, String> sEntry : hset) { // итератор по EntrySet  
        System.out.printf("%-24s = %s\n", sEntry.getKey(), sEntry.getValue());  
    }  
}
```

## Метод Адаптер

- Метод Адаптер используется для реализации интерфейса Iterable [lesson\\_ch11/ch/ex32/local](#)
  - адаптер добавляется как новый итератор в свой класс Iterable или в потомок Collection
  - **ВНИМАНИЕ.** Альтернативных итераторов может быть сколько угодно
- Синтаксис Адаптера
  - дальше автоматически создаются методы для реализации Iterator{}
  - тип <T> Generic заменяется на реальный тип в случае своего класса хранения

```
public Iterable<T> rev() {                                     // реализация альтернативного итератора
    return new Iterable<T>() {
        @Override
        public Iterator<T> iterator() {
            return new Iterator<T>() { }
```

## Custom Adapter

- Класс пользователя реализует интерфейс Iterable и может наследоваться от любого класса
  - в класс добавляется функция генератор Iterable () которая создает альтернативный Iterator
- Пример. свой класс альтернативный итератор

```
public class ShapeGen implements Iterable<Shape> {
    private Shape[] shapes;
    private int size;
    public ShapeGen(int size) {
        this.size = size;
        shapes = getShapes();
    }
    public Iterable<Shape> reversed() { // забабахали новый адаптер
        return new Iterable<Shape>() {
            private int index = shapes.length -1; // индекс реверсного итератора
            @Override
            public Iterator<Shape> iterator() {
                return new Iterator<Shape>() {
                    < реализация > ...
                }
            }
            @Override
            public Iterator iterator() {
                return new Iterator() {
                    < реализация > ...
                }
            }
        };
    }
}
```

## Collection Adapter

- Класс пользователя наследуются от Collection
  - затем добавляется функция генератор Iterable () которая создает альтернативный Iterator
  - таких альтернативных итераторов может быть сколько угодно
- Пример. Collection класс альтернативный итератор

```
public class ArrayListGen<T> extends ArrayList<T> { // Generic <T>
    public ArrayListGen(Collection<? extends T> c) { // конструктор коллекции
        super(c);
    }
    public Iterable<T> rev() { // реализация альтернативного итератора
        return new Iterable<T>() {
            @Override
            public Iterator<T> iterator() {
                return new Iterator<T>() {
                    < реализация > ...
                };
            }
        };
    }
}
```

## Выходы по Collections

### Array

- Массив это группа объектов одного типа
  - группа упорядочена по индексу
  - может хранить примитивы
  - размер массива фиксирован после создания

### Collection

- Collection это группа объектов разного типа
  - доступ методом перебора
  - Generic позволяют ограничить тип объектов хранения
  - примитивы может хранить в виде совместимых объектов
  - размер не фиксирован после создания

### Map

- Map это группа пар объектов <ключ, значение> разного типа
  - доступ методом перебора по списку <ключ>
  - не поддерживает Iterator нужна конвертация в Set(Map.Entryset)
  - остальное как у Collection
- HashMap потомок Map
  - быстрый доступ к элементам Map
- TreeMap потомок Map
  - автоматическая сортировка
  - требует у объекта хранения Comparable
- LinkedHashMap потомок Map
  - хранение объектов в порядке добавления

### List

- List это группа объектов разного типа
  - группа упорядочена по индексу
  - размер не фиксирован после создания
  - ArrayList потомок List быстрый произвольный доступ
  - LinkedList потомок List быстрый доступ к вставке объектов
- Stack потомок LinkedList
- Queue потомок LinkedList

### Set

- Set группа уникальных объектов разного типа
  - доступ методом перебора
  - остальное как у Collection
- HashSet быстрый доступ к элементам набора
- TreeSet упорядоченное хранение

### Не рекомендованы к использованию

- Vector
- HashTable
- Stack

## Exceptions

- Exception исключения, это параллельный основной программе процесс отработки ошибок
  - exception генерятся при помощи оператора throw
  - exception создаются в heap оператором new
  - управление после <throw exception> передается в ближайший обработчик exception
- Exception бывает разных типов
  - <Exception> общий тип реализует интерфейс Throwable его наследуют уточненные типы Exception
  - каждый тип можно обработать своим обработчиком
  - любой тип exception может обработать обработчик типа <Exception>
  - **ВНИМАНИЕ.** Так как Exception может перехватить любой exception обработчик ставят В КОНЕЦ
  -
- Синтаксис
  - try секция проверки на выполнение
  - catch секция перехвата исключений
  - finally выполняется всегда было исключение или нет
- ```
try { // блок проверки исключения
    throw new UnsupportedOperationException ("Message"); // сами генерим exception

} catch (UnknownFormatFlagsException e) { // блок проверки UnknownFormatFlags
    System.out.println(e.getMessage());

} catch (UnknownFormatConversionException e) { // блок проверки UnknownFormatConversion
    System.out.println(e.getMessage());

} catch (UnsupportedOperationException e) { // блок проверки UnsupportedOperationException
    System.out.println(e.getMessage());

} catch (Exception e) { // блок проверки Exception
    System.out.println(e.getMessage()); // сюда падают все неизвестные exception

} finally { // блок finally выполняется всегда
    System.out.println("Message from finally");
}
```

## Exception Interrupt and Renew

- Exception Interrupt
  - это использование Exception когда выполнение прерывается и передается в точку обработки
- Exception Renew
  - это использование Exception, когда перехват идет в точке возникновения проблемы
  - и код выполняется повторно до устранения проблемы в точке возникновения
- Пример. 10 раз повторяется процедура, пока проблема не будет исправлена, и нормальное продолжение

```
int k = 0;
boolean flag = true;
while (flag) {
    try {
        if (k < 10) {
            throw new Exception("Exception from while cycle");
        } else {
            flag = false; // выходим из цикла
        }
    } catch (Exception e) {
        System.out.println("Catch called :" + k++);
    }
}
System.out.println("Exception overridden by k:" + k);
```

## Custom Exception

- Custom Exception может быть класс пользователя, который наследует Exception [lesson\\_ch12/ch/ex1/local](#)
  - создать класс, который наследует Exception
  - создать функции с вызовом Custom Exception и throws Exception для обработки вне класса
  - создать конструктор Custom Exception Default и с параметром String для message()
  - **ВНИМАНИЕ.** для распечатки стека использовать e.printStackTrace()
- Пример Конструктор default и String

```
public class SimpleException extends Exception {  
}  
public class MyException extends Exception{  
    public MyException() {  
    }  
    public MyException(String message) {  
        super(message);  
    }  
}  
public class ClassWithException {      // класс методы которого порождают Exception  
    public void f() throws SimpleException {  
        System.out.print("f() calls SimpleException > ");  
        throw new SimpleException();      // нет сообщения, т.к. конструктор default  
    }  
    public void g() throws MyException {  
        System.out.print("g() calls MyException      > ");  
        throw new MyException("MyException Message");      // нет сообщения, т.к.  
конструктор default  
    }  
}  
public static void app() {  
    ClassWithException cwEx = new ClassWithException();  
    try {  
        cwEx.f();          // только после окружения блоком try/catch  
    } catch (SimpleException e) {  
        e.printStackTrace();      // только после окружения блоком try/catch  
    }  
    try {  
        cwEx.g();          // только после окружения блоком try/catch  
    } catch (MyException e) {  
        System.out.println(e.getMessage());  
        e.printStackTrace();  
    }  
}
```

## Error

- Error это еще один вид исключений, но он обрабатывается независимо от Exception
- Error бывает разных типов
  - <Error> это более серьезный вид ошибок в программе отрабатывается НЕЗАВИСИМО от Exception
  - Error отрабатывает все виды Error
  - **ВНИМАНИЕ.** Так как Error отрабатывает все типы error, обработчик ставят В КОНЕЦ

## Logging

- Регистрацию Exception можно встроить в Custom Exception Class или использовать Logger External
- Logger in Custom Exception Class
  - создается класс Exception пользователя
  - в конструктор добавляется информация для Logger и запись в Logger
- Пример.
  - для заполнения сообщения стека используется класс StringWriter но можно и свое сообщение
- ```
public class LoggingException extends Exception {
    private static Logger logger = Logger.getLogger("Logger.Name"); // custom logger
    public LoggingException() {
        StringWriter stringWriter = new StringWriter(); // экземпляр класса StringWriter
        printStackTrace(new PrintWriter(stringWriter)); // заполнить вызов стека текст
        logger.severe(stringWriter.toString()); // записать в log и на печать
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName(); // для вывода просто имени класса
    }
}
public class Local {
    public static void app() {
        try {
            throw new LoggingException(); // Exception with Logger
        } catch (LoggingException e) {
            System.err.println("Catched: " + e); // вывод собственно объекта
        }
    }
}
```

[lesson\\_ch12/ch/ex6/local](#)

## Logger External

- Регистрация исключений при помощи внешнего logger который обрабатывает Exceptions
- Пример. обработка разных исключений и запись в Log
  - несмотря на восходящее преобразование exceptions в log идут оригинальные

[lesson\\_ch12/ch/ex6/local](#)

```
public class LoggingExternal {
    private Logger logger = Logger.getLogger("Logger.Name"); // custom logger
    public void getLog(Exception e) {
        StringWriter stringWriter = new StringWriter(); // экземпляр StringWriter
        e.printStackTrace(new PrintWriter(stringWriter)); // заполнить стек Exception
        logger.severe(stringWriter.toString()); // записать в log и на печать
    }
}
public class Local {
    public static void app() {
        LoggingExternal logger = new LoggingExternal();
        try {
            throw new NullPointerException();
        } catch (Exception e) {
            logger.getLog(e); // сохранить в Logger
            System.err.println(logger);
        }
        try {
            throw new UnsupportedOperationException();
        } catch (Exception e) {
            logger.getLog(e); // сохранить в Logger
        }
    }
}
```

## Спецификация исключений

- При описании метода можно сообщить о порождаемых им исключениях
- Исключения бывают “checked” и “unchecked”
  - “checked” выдают ошибку компиляции, если нет обработки, это часть потомков класса Exception
  - “unchecked” не выдают ошибку, это часть Exception, RuntimeExceptions и Error
- **ВНИМАНИЕ.** При описании метода РЕКОМЕНДУЕТСЯ сообщать об исключениях которые порождает метод
- Синтаксис
  - Имена порождаемых Exception пишутся в названии функции

```
void f() throws LastOwnerException, MimeTypeParseException, UnknownHostException {  
}
```

## Методы Throwable

- Методы позволяют получить больше информации об исключении [lesson\\_ch12/ch/ex9/local/include](#)
- 
- getMessage() выдать сообщение в строку
- getLocalizedMessage() выдать сообщение в строку
- toString() стандартная функция вывода Exception в String, есть в базовом классе
- printStackTrace() распечатывает полный стек Exception [lesson\\_ch12/ch/ex10/include](#)
- fillinStackTrace() распечатывает частный стек Exception [lesson\\_ch12/ch/ex10/include](#)
- getClass() полное имя класса exception
- getClass().getSimpleName() сокращенное имя класса exception

## Трассировка стека программ

- Метод getStackTrace() выдает массив кадров стека [lesson\\_ch12/ch/ex10/local](#)
  - каждый кадр это объект который обладает кучей методов
  - можно прямо в блоке catch перебрать весь стек кадров методом foreach
- Пример [lesson\\_ch12/ch/ex10/local](#)

## Разница между printstackTrace и fillinStackTrace

- printStackTrace() распечатывает весь массив кадров стека
  - **ВНИМАНИЕ.** чтобы не мешался вывод на печать можно подменить вывод printStackTrace с err на out
- ```
try {  
    f();  
} catch (Exception e) {  
    e.printStackTrace(System.out); //распечатка стека Exception в System.out  
}
```
- fillinStackTrace() вытаскивает массив кадров стека для текущей точки перехвата Exception
  - Разница в том, что [lesson\\_ch12/ch/ex10/include](#)
    - при распечатке printStackTrace везде будет один и тот же полный стек кадров
    - при распечатке fillinStackTrace в разных местах будет разный стек кадров согласно структуре софта
  - **ВНИМАНИЕ.** fillinStackTrace() это ОБРАЗ СТЕКА в ТЕКУЩЕЙ ТОЧКЕ можно использовать как
  - новый адрес для генерации exception [lesson\\_ch12/ch/ex10/include](#)

```
public static void h() throws Exception {  
    try {  
        f();  
    } catch (Exception e) {  
        e.printStackTrace(System.out);  
        throw (Exception)e.fillInStackTrace(); //создание новой точки Exception  
    }  
}
```

## Повторное возбуждение исключения

- Повторное возбуждение передает exception на верхний уровень
  - Повторное возбуждение это вызов <throw eX> в блоке catch обработки этого самого <eX>
- Пример.

```
public class Integra {    public static void f() throws Exception {        throw new Exception("Generated in f()");    }    public static void g() throws Exception {        try {            f();        } catch (Exception e) {            throw e;        }    }    public static void app() {        try {            g();        } catch (Exception e) {            e.printStackTrace(System.out);        }    }}lesson\_ch12/ch/ex10/integra
```
- Повторное возбуждение, создание новой точки происхождения
  - по сути перехватывается Exception и отправляется как вновь созданное из текущего места
  - fillInStackTrace() используется для создания нового контекста при создании Exception
- Пример

```
public class Integra {    public static void f() throws Exception {        throw new Exception("Generated in f()");    }    public static void h() throws Exception {        try {            f();        } catch (Exception e) {            throw (Exception)e.fillInStackTrace(); //создание новой точки Exception        }    }    public static void app() {        try {            h();        } catch (Exception e) {            e.printStackTrace(System.out); //контекст распечатается как для новой точки        }    }}lesson\_ch12/ch/ex10/integra
```
- Повторное возбуждение Exception разного типа
  - перехватывается exception одного типа, а внутри catch возбуждается другого типа
- **ВНИМАНИЕ.** при перехвате Ex одного типа и подмене его на другое, контекст первого теряется
- Пример

```
public class RetryNewEx {    private void f() throws MEx1 {        throw new MEx1("f() . MEx1_message"); // генерация исходного exception MEx1    }    private void g() throws MEx2 {        try {            f();        } catch (MEx1 e) {            throw new MEx2("g() . MEx2_message"); // перехват исходного exception MEx1 // генерация нового exception MEx2        }    }    public void retryException() {        try {            g();        } catch (MEx2 e) {            e.printStackTrace(System.out); //перехват нового exception MEx2        }    }}lesson\_ch12/ch/ex10/retry
```

## Цепочки исключений

- Цепочки исключений это возбуждение одного исключения другим с сохранением информации
  - цепочки исключений принимают в качестве аргумента конструктора объект Exception
  - конструктор поддерживает три класса Throwable      Exception, Error, RuntimeException
  - метод initCause() вместо конструктора                  все остальные классы Exception
- При срабатывании Exception с Cause внутри работает так
  - срабатывает catch Exception «носителя»
  - в распечатке стека покажет Exception «cause» и номер строки, где cause был заложен в носителя
- **ВНИМАНИЕ.** Сработает catch DynamiFieldsException в распечатке стека будет 2nd с NullPointerException
- Пример. 1<sup>st</sup> строка где сработал catch носителя, строка 2<sup>nd</sup> где заложен cause [lesson\\_ch12/ch/ex10/chain](#)

```
public Object setField(String key, Object value) throws DynamicFieldsException {  
    if (value == null) {  
        DynamicFieldsException dfe = new DynamicFieldsException(); // 1st exception  
        dfe.initCause(new NullPointerException()); // 2nd cause Ex  
        throw dfe; // выбрасываем Custom Exception с источником внутри  
    }  
}
```

•

## Стандартные исключения Java

- Все исключения Error, Exception, RunTimeException базируются на интерфейсе Throwable
  - Error                                        системные ошибки и ошибки компиляции
  - Exception                                    исключения любого вида
  - RuntimeException                            исключение во время исполнения
- Свойства исключений
  - имя исключения описывает ситуацию в которой возникло исключение
  - библиотеки исключений      java.lang, java.util, java.net, java.io

## RuntimeException

- RuntimeException это исключение исполнения
  - отрабатывает группу автоматических проверок во время исполнения
  - является unchecked то есть не требует блока try для окружения
  - так как это exception ошибок программиста по сути, если программист его не перехватил,
  - то RuntimeException вылетает в main() и распечатывает стек программы
- **ВНИМАНИЕ.** Если RuntimeException добирается до main() и распечатывает стек программы

## Finally

- Finally нужен для гарантированного ОСВОБОЖДЕНИЯ РЕСУРСОВ, НЕ ПАМЯТИ, сетевых, дисковых ресурсов
- Finally блок выполняется всегда, независимо от того, было Exception или нет [lesson\\_ch12/ch/ex1/local](#)
  - Finally нужен для выполнения независимо было возбуждено или нет exception внутри try()
  - Finally может быть прерван, только если до него выполнен System.Exit(0)
- **ВНИМАНИЕ.** Finally позволяет отработать нештатную ситуацию при любом раскладе вызовов Exception
- Синтаксис

```
try {
    f(i);
} catch (LastOwnerException e) {
    e.printStackTrace();
} catch (MimeTypeParseException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    System.out.println("finally: done");
}
```

## Finally выполнение

- Finally выполняется всегда
  - даже если внутри блока try произошло исключение или возврат
- Пример [lesson\\_ch12/ch/ex12a/include](#)

```
while (true) {
    try {
        if (count++ == 0) {
            throw new ThreeEx();
        }
    } catch (ThreeEx e) { // срабатывает только когда есть исключение

    } finally {
        if (count == 2) {
            break;
        }
    }
}
```

## Finally освобождение ресурсов

- Finally позволяет освободить или вернуть в исходное любые ресурсы [lesson\\_ch12a/ch/ex12/integra](#)
  - finally не предназначен только для освобождения памяти, памятью занимается GC
  - finally позволяет отработать освобождение более компактно
- Пример. освобождение ресурса Switch.off() без finally
  - освобождение ресурсов здесь производится в основном блоке try и в каждом блоке catch
  -

```
public static void app() {
    Switch sw = new Switch();
    try {
        sw.on();
        f(); // генератор Exceptions SwEx1, SwEx2
        sw.off();
    } catch (SwEx1 e) {
        System.out.print("catch:" + e.getClass().getSimpleName() + ":");
        sw.off();

    } catch (SwEx2 e) {
        System.out.print("catch:" + e.getClass().getSimpleName() + ":");
        sw.off();
    }
}
```

## Finally и Сквозные Exception

- Finally и Сквозные Exception
  - если в блоке try возбуждается «сквозное» exception которое не отрабатывается блоками catch
  - сначала отрабатывается finally, затем отрабатываются catch верхнего уровня
- Пример

```
public static void app() {  
    // w finally  
    System.out.println("Try_Catch_w_Finally:");  
    try {  
        try {  
            sw.on();  
            f(); // генератор exceptions 1,2,4 уровня  
        } catch (SwEx1 e) { // отработает SwEx1  
            System.out.println("catch: SwEx1");  
        } catch (SwEx2 e) { // отработает SwEx1  
            System.out.println("catch: SwEx2");  
        } finally {  
            System.out.print("finally:");  
            sw.off(); // отработает finally и для SwEx4  
        }  
    } catch (SwEx4 e) { // сюда после finally хотя это catch  
        System.out.println("catch: SwEx4");  
    } finally {  
        System.out.println("finally2:");  
    }  
}
```

[lesson\\_ch12/ch/ex12a/integra](#)

## Finally и операторы continue и break

- Finally также выполняется до перехвата управления continue; и break;
- Вместе с labels он позволяет заменить goto ???
- 
- 

## Finally and Lost Exception

- Finally может потерять Exception когда одно Exception замещается другим [lesson\\_ch12/ch/ex18/local](#)
- Finally может потерять Exception когда в блоке стоит return
- Защита от потерь Finally Exceptions это поставить персональные CATCH [lesson\\_ch12/ch/ex18/access](#)
- Пример [lesson\\_ch12/ch/ex19/access](#)

```
public static void app() {  
    try {  
        try {  
            f(); // Способ 1 замещение  
        } finally {  
            dispose(); // здесь генерится STDEException  
        }  
    } catch (Exception e) {  
        System.out.println(e); // видим только STDEException потеря VIPEException  
    }  
    try {  
        try {  
            f(); // Способ 2 использование return в finally  
        } finally {  
            return; // здесь генерится VIPEException  
        }  
    } catch (Exception e) {  
        System.out.println(e); // отрубает все исключения при возврате  
    }  
}
```

## Finally и Return

- Finally может отработать корректный возврат из функции, при вызове return откуда угодно внутри try
  - управление прямо перед возвратом будет отдано finally
- ```
public static void f() {
    CADSystem cad = new CADSystem(15);
    try {
        return;
    } catch (Exception e) {

    }finally {
        cad.dispose(); // по любому отрубить
    }
}
```

## Finally и Return(value)

- Finally блок работает особым образом с переменными return(value)
- Finally может быть не выполнен только при выполнении System.Exit(0)
- **ВНИМАНИЕ.** finally работает ОСОБЫМ ОБРАЗОМ в try\_catch если там есть return(value)
- try return(value) побеждает finally value = new value
- catch return(value) побеждает finally value = new value
- finally return(value) побеждает try return(value), catch return(value)
- Пример.

```
public static int f() {
    int i = 1;
    try {
        i = 2;
        return i;
    } catch (Exception e) {
        i=3;
        return i;
    }finally {
        i = 5; // да здесь будет изменено значение, но i уже в стеке try
    }
}
public static int g() {
    int i = 1;
    try {
        i = 2;
        throw new Exception();
    } catch (Exception e) {
        i=3;
        return i;
    }finally {
        i = 5; // да здесь будет изменено значение, но i уже в стеке try
    }
}
public static int h() {
    int i = 1;
    try {
        i = 2;
        throw new Exception();
    } catch (Exception e) {
        i=3;
        return i;
    }finally {
        i = 5;
        return i;
    }
}
```

## Ограничения Exceptions от Exception

- Exceptions имеют следующие ограничения при наследовании классов или реализации интерфейсов
  - конструктор потомка ОБЯЗАН возбуждать исключения конструктора предка
  - конструктор класса потомка может возбуждать любые НОВЫЕ исключения
  - переопределенный метод НЕ МОЖЕТ возбуждать новые Exceptions
  - переопределенный метод МОЖЕТ возбуждать Exceptions или потомков Exceptions метода предка
  - переопределенный метод МОЖЕТ не возбуждать Exceptions

## Исключения из правил RuntimeException

- ВНИМАНИЕ.** если переопределенный метод заявлен в разных классах с разными Exceptions
  - метод МОЖЕТ не возбуждать Exceptions и перехватить от предка внутри себя
- ВНИМАНИЕ.** если Exception от RuntimeException то все ограничения снимаются
- не надо try\_catch
- не надо прописывать throws на выходе методов
- можно возбуждать в потомках любые комплекты исключений
- Пример

[lesson\\_ch12/ch/ex20/local](#)

```
public class StormyInning extends Inning implements Storm {  
    public StormyInning() throws BaseBallEx, RainedOut, Light { //базовая и дополнительная  
        super(); // так как super() первый элемент, то некуда вставить try{}  
    }  
    //    public void walk() throws PopFoul { //нельзя возбуждать новые исключения в потомке  
    //                                // если их нет в базовой версии  
    //    }  
    public StormyInning(String s) throws BaseBallEx, Foul{ // базовая и дополнительная  
    }  
    @Override  
    public void rainHard() throws RainedOut { // базовые версии  
    }  
    @Override  
    public void atBat() throws Strike, PopFoul, Light { // базовые версии и потомки  
    }  
    public void event() { // единственный вариант, когда два предка с разными исключениями  
    }  
}
```

- Перехват и ограничения

```
public class Local {  
    private static void cn(Exception e) {  
        System.out.println(e.getClass().getSimpleName());  
    }  
    public static void app() {  
        try { // класс ПОТОМОК  
            StormyInning si = new StormyInning();  
            si.atBat();  
            si.rainHard();  
            si.event(); // переопределенный метод ничего не вызывает  
        } catch (PopFoul e) { // atBat  
        } catch (RainedOut e) { // конструктор, rainHard  
            cn(e);  
        } catch (BaseBallEx baseBallEx) { //Light, Strike ловим по предку  
            baseBallEx.printStackTrace();  
        }  
        try { // восходящее преобразование  
            Inning i = new StormyInning("str");  
            i.atBat();  
            i.event(); // переопределенный метод ничего не вызывает  
        } catch (Foul e) { // atBat PopFoul ловим по предку  
        } catch (BaseBallEx e) { // конструктор, atBat Strike, Foul  
        }  
    }  
}
```

## Конструктор Exception

- Конструкторы особый случай, если в конструкторе генерируется Exception [lesson\\_ch12/ch/ex21/local](#)
  - если конструктор не завершен то при new Exception надо закрыть то, что еще не создано
  - **ВНИМАНИЕ.** new Exception внутри конструктора делает ситуацию НЕПРЕДСКАЗУЕМОЙ с ресурсами
  - **ВНИМАНИЕ.** Закрытие ресурсов dispose() ВСЕГДА вызывать ВРУЧНЮЮ и НЕ привязывать к finalize()

## Метод 1. Выделенная обработка Exception

- В конструкторе есть ключевые ситуации, которые надо отрабатывать отдельно
- Как пример, при открытии файла ситуация 1 файл открыт, ситуация 2 файл не открыт
  - ситуация 1 файл открыт, все исключения в этом случае должны сначала закрыть файл
  - ситуация 2 файл не открыт, все исключения просто идут мимо
- **ВНИМАНИЕ.** Закрытие ресурсов dispose() ВСЕГДА вызывать ВРУЧНЮЮ и НЕ привязывать к finalize()
- Общий принцип. Выделить ключевые ситуации.
- Пример. корректного закрытия файла при открытии в конструкторе

```
public class InputFile {  
    private BufferedReader inFile;  
    private String fName;  
  
    public InputFile(String fName) throws Exception {  
        this.fName = fName;  
        try {  
            inFile = new BufferedReader(new FileReader(fName)); // открыли файл  
  
        } catch (FileNotFoundException e) {  
            System.out.println("File Not Found: " + fName);  
            throw e; // выбрасываемся на верхний уровень из программы  
        } catch (Exception e) { // все остальные исключения выполняем закрытие файла  
            try {  
                inFile.close(); // пытаемся закрыть файл  
            } catch (IOException e2) { // e2 потому что надо сохранить e  
                System.out.println("File Close Error.Constructor");  
            }  
            throw e; // после закрытия файла выбрасываемся  
        } finally {  
            System.out.println("Finally: No Close File Here");  
        }  
    }  
    public String getLine() {  
        String s;  
        try {  
            s = inFile.readLine();  
        } catch (IOException e) {  
            throw new RuntimeException("Read File Error.readline()");  
        }  
        return s;  
    }  
    public void dispose() { // это запускать все равно вручную  
        try {  
            inFile.close();  
            System.out.println("File Closed");  
        } catch (IOException e) {  
            throw new RuntimeException("File Close Error.dispose()");  
        }  
    }  
}
```

## Метод 2. Вложенные блоки Try

- Вложенные блоки в конструкторе это самый безопасный способ
  - основной принцип это блок try\_finally{} сразу после создания объекта
- **ВНИМАНИЕ.** Каждый объект требующий завершения ДОЛЖЕН иметь try\_finally сразу после создания
- Методика
  - создается Constructor который безопасно работает с Exception и генерит новые наверх
  - во внешней программе создание экземпляра окружается блоком try{}
  - сразу после создания экземпляра создается второй блок try{} внутри которого основная работа
  - внутренний цикл завершается finally{} который закрывает объект всегда
  - **ВНИМАНИЕ.** в случае ошибок конструктора вылетает во внешний catch()
  - в случае ошибок в работе вылетает во внутренний catch() и отработает finally{}

### Пример.

```
public class Include {
    public static void app() {
        try {
            InputFile f = new InputFile("src/ch/ex21/local/Local.java");
            try {
                String s;
                int i = 1;
                while ((s = f.getLine()) != null) { // распечатка строк в цикле
                    System.out.printf("%3d:%s\n", i++, s);
                }
            } catch (Exception e) { // внутренний цикл работы с Exception файл открыт
                System.out.println("Inner Exception:");
            } finally { // исключение при открытом экземпляре или штатный выход
                f.dispose();
            }
        } catch (Exception e) { // внешний цикл работы с Exception файл не открыт или
            System.out.println("External Exception:");
        }
    }
}
```

## Правила создания вложенных блоков Try

- Классы можно разделить на две группы
  - Тип 1 Не создает ошибок при конструировании
  - Тип 2 Создает ошибки при конструировании
- Методика для класса типа 1 без ошибок
  - Создать объект в обычном режиме
  - Исполнение подключить блок try\_finally после создания
- Группировка объектов типа 1
  - Создать объекты в штатном режиме
  - Исполнение в блоке try\_finally один блок на все объекты
  - Закрытие объектов в блоке finally в обратном порядке конструирования
- Методика для класса типа 2 с ошибками
  - Подключить блок try\_catch при создании объекта
  - Подключить блок try\_finally внутри try\_catch сразу после создания
- Группировка объектов
  - Последовательно создавать вложенные блоки try\_catch/try\_finally на каждый объект
  - Подключать блоки try\_finally на каждый объект свой блок
  - Выстроить иерархию в виде матрешки try{nc1 try{ nc2 try{ nc3} finally nc3 } finally nc2 } finally nc1}

## Exceptions Matching

- Совмещение Exceptions
  - это механизм поиска и запуска «примерного» обработчика для Exceptions
  - то есть обработчик для базового класса Custom Exception может отработать и всех его потомков
- Пример.

```
public class Annoy extends Exception {  
}  
public class Sneeze extends Annoy {  
}  
public class Local {  
    public static void app() {  
        // точный перехват  
        try {  
            throw new Sneeze();  
        } catch (Sneeze e) { // точная отработка сработает тут  
            System.out.println("catch:"+e.getClass().getSimpleName());  
        } catch (Annoy e){ // сюда не дойдет  
            System.out.println("catch1:"+e.getClass().getSimpleName());  
        }  
        // примерный перехват  
        try {  
            throw new Sneeze();  
        } catch (Annoy e){ // Sneeze отработает здесь по matching как потомка  
            System.out.println("catch2:"+e.getClass().getSimpleName());  
        }  
    }  
}
```

[lesson\\_ch12/ch/ex25/local](#)

## Альтернативные решения

- Система отработки исключений это механизм обхода мест где метод не справляется
- Главное правило НЕ отрабатывать исключение пока неизвестно что делать
- **ВНИМАНИЕ.** НЕЛЬЗЯ поглощать Exception на месте, иначе оно теряется, лучше отправить его «наверх»
- Контролируемое исключение
  - это исключение которое перехвачено блоком catch на месте его возникновения
  - «низвержение исключений» это поглощение исключений на местах где неясно как обработать
- Неконтролируемое исключение
  - это исключение которое «подымается» по иерархии вверх до уровня где будет обработано
- **ВНИМАНИЕ.** Неконтролируемые исключения это наилучший способ работы с исключениями
- Пример. отправка Exception на консоль

[lesson\\_ch12/ch/ex26/access](#)

```
public class Access {  
    public static void app() throws Exception{ // отправка Exception на консоль  
        FileInputStream f = new FileInputStream("Main.java");  
        f.close();  
    }  
}
```

## Конвертация Контролируемых исключений в Неконтролируемые

- Для перевода Exception из контролируемого в неконтролируемое
  - надо в блоке перехвата УПАКОВАТЬ Exception в Runtime Exception
- **ВНИМАНИЕ.** RuntimeException является unchecked и позволяет протащить Exception наверх
- **ВНИМАНИЕ.** RuntimeException unchecked нет ограничений на наследуемые методы [ch12/ch/ex29/access](#)
- **ВНИМАНИЕ.** RuntimeException не надо прописывать на выходе методов [lesson\\_ch12/ch/ex30/access](#)
- Пример

```
public class WrapRTE {  
    void throwRTE(int type) { // ВНИМАНИЕ не надо прописывать выходные Exceptions  
        try {  
            switch (type) {  
                case 0: throw new FileNotFoundException(); // обработчик 1  
                case 1: throw new FileNotFoundException();  
                case 2: throw new FileNotFoundException();  
                default: return; // ВНИМАНИЕ нет exception  
            }  
        } catch (Exception e) {  
            throw new RuntimeException(e); // unchecked Exception отправляем наверх  
        }  
    }  
}  
  
public class Local {  
    public static void app() {  
        WrapRTE rte = new WrapRTE();  
        for (int i = 0; i < 4; i++) {  
            try {  
                if (i < 3) {  
                    rte.throwRTE(i);  
                }  
            } catch (RuntimeException re) { // отрабатываем контейнер RT  
                try { // можно switch, можно try_catch  
                    System.out.println(re.getCause()); // это исходное Exception  
                    throw re.getCause(); // запускаем исходное Exception  
                } catch (FileNotFoundException e) { // обработчик 1  
                } catch (IOException e) { // обработчик 2  
                } catch (ClassCastException e) { // обработчик 3  
                } catch (Throwable throwable) { // это требование среды для getCause()  
                }  
            }  
        }  
    }  
}
```

- Пример прямой упаковки Демо

[lesson\\_ch12/ch/ex27/local](#)

```
public class Access {  
    public static void app() {  
        int[] ints = new int[]{0, 1, 2, 3, 4};  
        try {  
            try {  
                ints[12] = 12;  
            } catch (IndexOutOfBoundsException e) {  
                throw new RuntimeException(e); // отправляем наверх в упаковке  
            }  
            System.out.println(Arrays.toString(ints));  
        } catch (RuntimeException rt) { // поймали наверху  
            try {  
                throw rt.getCause(); // распаковали и запускаем  
            } catch (IndexOutOfBoundsException e) { // перехватили исходное exception  
                System.out.println("catch: runtime:"+e.getClass().getSimpleName());  
            } catch (Throwable e) {  
                System.out.println("catch: throwable:"+e.getClass().getSimpleName());  
            }  
        }  
    }  
}
```

## RuntimeException Предостережения

- RuntimeException является «unchecked» работает абсолютно прозрачно [lesson\\_ch12/ch/ex27/local](#)
- Вся работа по контролю и отлову RuntimeException на программисте [lesson\\_ch12/ch/ex29/access](#)
- Любое Exception которое наследует RuntimeException также прозрачно [lesson\\_ch12/ch/ex30/access](#)

## Методы

- **ВНИМАНИЕ.** RuntimeException не надо прописывать на выходе методов
- **ВНИМАНИЕ.** RuntimeException является unchecked и позволяет протащить Exception наверх
- **ВНИМАНИЕ.** если Exception от RuntimeException то все ограничения снимаются
  - не надо try\_catch
  - не надо прописывать throws на выходе методов
  - можно возбуждать в потомках любые комплекты исключений

## Наследование

- **ВНИМАНИЕ.** RuntimeException unchecked нет ограничений на наследуемые методы
- **ВНИМАНИЕ.** если переопределенный метод заявлен в разных классах с разными Exceptions
  - метод МОЖЕТ не возбуждать Exceptions и перехватить от предка внутри себя

## Использование RuntimeException

- Однозначно рекомендуется использовать RuntimeException везде где только можно
- они позволяют сосредоточиться на разработке кода

## Применение Exceptions

- Всегда стараться применять Exceptions для того чтобы
- Обработка на текущем уровне
  - обработать ошибку на текущем уровне,
  - если невозможно обработать на текущем, НЕ ПЕРЕХВАТИВАТЬ пока не станет ясно как обработать
- Исправление проблемы и повторный вызов
  - исправить проблему и вызвать метод, возбудивший Exception
- Исправление проблемы без повторного вызова
  - сделать все для исправления Exception на месте возникновения, продолжить выполнение и
  - избежать повторный вызов метода создавшего Exception
- Поиск альтернативы
  - попытаться найти альтернативный результат, вместо того, который должен был выдать метод
- Проброс наверх
  - попытаться исправить в текущем контексте, и если не вышло, заново возбудить это же Exception
  - выбросить его на верхние уровни
- Завершение программы
  - перехватить и завершить работу программы
- Упрощение кода
  - упростить код программы, применять Exception только для упрощения кода.
- Повышение безопасности
  - повысить уровень безопасности и надежности кода.

## Strings Строки

- Strings самый часто используемый класс в программах
- String являются immutable, методы String всегда создают НОВЫЙ объект с изменениями

[lesson\\_ch12a/ch/ex1/local](#)

## StringBuilder и перегруженный оператор String “+”

- При сложении строк « +» всегда автоматически создается новый объект StringBuilder в цикле
- **ВНИМАНИЕ.** Независимо где расположен, оператор «+» для строке всегда создает НОВЫЙ StringBuilder  
ВСЕГДА ИЗБЕГАТЬ в ЦИКЛЕ оператор «+» для СТРОК
- StringBuilder созданный вручную и цикл добавления это максимально быстрая и компактная реализация
- Пример.

[lesson\\_ch12a/ch/ex1/include](#)

[lesson\\_ch12a/ch/ex1/integra](#)

- **ВНИМАНИЕ.** В ЦИКЛЕ использовать StringBuilder
- В ОДНОЙ строке НАДО использовать String «+»

```
• Цикл где надо использовать StringBuilder
• Пример.
• public class Integra {
    public static void app() {
        // в цикле ЛУЧШИЙ СПОСОБ
        System.out.println("В цикле ЛУЧШИЙ СПОСОБ");
        StringBuilder sb;
        sb = new StringBuilder("[");
        for (int i = 0; i < 25; i++) {
            sb.append(Range.getInt(10)); // добавляем строку
            sb.append(", ");
        }
        sb.delete(sb.length() - 2, sb.length()); // подрезаем строку
        sb.append("]");
        System.out.println(sb.toString());
        // в цикле САМЫЙ РАСТОЧИТЕЛЬНЫЙ И МЕДЛЕННЫЙ СПОСОБ
        System.out.println("В цикле САМЫЙ РАСТОЧИТЕЛЬНЫЙ И МЕДЛЕННЫЙ СПОСОБ");
        String s = "[";
        for (int i = 0; i < 25; i++) {
            s += Range.getInt(10) + ", ";
        }
        s = s.substring(0, s.length() - 2);
        s += "]";
        System.out.println(s);
    }
}
```

## Анализ ByteCode String

- Java анализ и декомпиляция кода
  - зайти в src
  - скомпилировать код javac ch/ex1/Code.java
  - декомпилировать код javap -c ch.ex1.Code
  - проанализировать результат декомпиляции
- Пример.
- src>javac ch/ex1/analyze/Concatenation.java // компиляция кода
- src > javap -c ch.ex1.analyze.Concatenation // декомпиляция кода
- Результат декомпиляции

Code:

```
0: ldc          #2           // String mango
2: astore_1
3: new          #3           // class StringBuilder
6: dup
7: invokespecial #4          // Method StringBuilder."<init>":()V
10: ldc         #5           // String abc
```

## Анализ кода Integra

- анализ кода integra
    - на использование ресурсов StringBuilder и String
  - Процедура анализа
    - создать проект
    - скомпилировать и запустить
    - зайти в Terminal > перейти по пути > lesson\_ch12a/out/production/lesson\_ch12a
    - декомпилировать код > javap -c ch.ex1.integra.Integra
  - Результат декомпиляции
  - Результат анализа      в данном случае в каждом цикле String создается объект StringBuilder
  - Вывод                    применение String «+» неприемлемо, надо делать на StringBuilder
- Code:
- ```
// StringBuilder Version
 24: new      #7          // class StringBuilder новый объект
 28: ldc       #8          // String [
 36:          // цикл на 65 строку >>
 45: invokestatic #10        // Method lib/utils/Range.getInt:(I)I
 48: invokevirtual #11        // Method StringBuilder.append
 53: ldc       #12         // String ,
 55: invokevirtual #13        // Method StringBuilder.append
 62: goto       36          // << цикл на 36 строку
 81: ldc       #16         // String ]
 83: invokevirtual #13        // Method StringBuilder.append
 91: invokevirtual #17        // Method StringBuilder.toString
 94: invokevirtual #4          // Method java/io/PrintStream.println

// String Version
105: ldc       #8          // String [
113: if_icmpge 150         // переход на строку 150 >>
116: new      #7          // class StringBuilder новый объект в итерации
124: invokevirtual #13        // Method StringBuilder.append:(
129: invokestatic #10        // Method lib/utils/Range.getInt:(I)I
132: invokevirtual #11        // Method StringBuilder.append
135: ldc       #12         // String ","
137: invokevirtual #13        // Method StringBuilder.append:;
140: invokevirtual #17        // Method StringBuilder.toString():LString;
147: goto       110         // << цикл на 110 строку
158: invokevirtual #21        // Method String.substring:(II)LString;
162: new      #7          // class StringBuilder      новый объект
170: invokevirtual #13        // Method StringBuilder.append
173: ldc       #16         // String ]
175: invokevirtual #13        // Method StringBuilder.append
178: invokevirtual #17        // Method StringBuilder.toString():LString;
186: invokevirtual #4          // Method java/io/PrintStream.println
189: return
```

[lesson\\_ch12a/ch/ex1/integra/](#)

## Анализ кода SprinklerSystem

- Анализ кода SprinklerSystem метод toString
- Цель анализа определить есть ли множественное создание объектов StringBuilder в цикле
- Вывод                    в toString создается всего один объект и функция работает компактно и быстро.

public java.lang.String toString();

Code:

```
 0: new      #5          // class java/lang/StringBuilder
 7: ldc       #7          // String valve1='
 9: invokevirtual #8          // Method java/lang/StringBuilder.append
 ...
 93: getfield   #20         // Field i:I
105: getfield   #23         // Field f:F
108: invokevirtual #24        // Method java/lang/StringBuilder.append
111: invokevirtual #25        // Method java/lang/StringBuilder.toString
114: areturn
```

[lesson\\_ch12a/ch/ex1/access/access2](#)

## Непреднамеренная рекурсия

- Непреднамеренная рекурсия когда в метод `toString()` вставляют «`this`»  
  - механизм такой компилятор заходит в `this.toString()` для данного объекта
  - просчитывает строку «`Local{}`» затем «`+`» и далее встречает «`this`»
  - «`this`» это объект поэтому вызывается `toString.this()`, а мы уже внутри `toString()` то есть рекурсия
  - так как это рекурсия внутри команды `<return "Local"+this>` прервать это невозможно
- Решение  
[lesson\\_ch12a/ch/ex2/local](#)
  - для вызова объекта `this` внутри `toString()` без рекурсии надо вызывать `super.toString()`
  - в результате генерится ссылка объекта, которая и вставляется с строку `toString`
- Пример.

```
public class Access {  
    @Override  
    public String toString() {  
        // return "Local{}"+this;           // место возникновения рекурсии  
        return "Access{}"+super.toString(); // исправленная версия  
    }                                         // super.toString() это фиксированная строка  
  
    public static void app() {  
        Access access = new Access();  
        String s = access.toString();     // место возникновения рекурсии  
        System.out.println(s);  
    }  
}
```

## Операции со строками

- Constructor  
[lesson\\_ch12a/ch/ex3/local](#)
- `length()` длина строки
- `charAt()` char на позиции
- `getChars()` массив `char[]` подстроки
- `getBytes` массив `byte[]` Charset это кодировка
- `toCharArray()` массив `char[]`
- `equals()` сравнение строк по значению равно/не равно
- `equalsIgnoreCase()`
- `compareTo()` сравнение строк компаратор -1, 0, 1
- `contains()` содержит ли подстроку
- `contentEquals()` сравнение на равенство всей строки
- `regionMatches()` сравнение подстрок
- `startsWith()` сравнение начала строки и подстроки
- `endsWith()` сравнение конца строки и подстроки
- `indexOf()` индекс символа в подстроке
- `substring()` подстрока
- `subSequence()` подстрока
- `concat()` соединение строк
- `replace()` замещение строки
- `toLowerCase()` в нижний регистр
- `toUpperCase()` в верхний регистр
- `trim()` сжать пробелы в начале и конце строки
- `valueOf()` перевести primitive в строку
- `intern()` смотреть пример

## Форматирование вывода

- `String.format()` использование шаблона форматирования
  - шаблон форматирования позволяет вывести строки, символы и числа в заданном формате
  - вывод осуществляется функциями `System.out.format()` и `System.out.printf()` они одинаковые

## Formatter Class

- `Formatter` это форматированный вывод в заданный поток на печать
- в экземпляре задается в конструкторе поток вывода
  - в экземпляре в методах в каком формате выводить данные
- Пример. два экземпляра выводятся в заданном формате в разные потоки
- ```
public class Turtle {  
    private String name;  
    private Formatter f;  
    public Turtle(String name, Formatter f) {  
        this.name = name;  
        this.f = f;  
    }  
    public void move(int x, int y) {  
        f.format("%s The Turtle is at (%4d,%-4d)\n", name, x, y);  
    }  
}  
  
public class Access {  
    public static void app() {  
        System.out.println("\n====ACCESS==");  
        System.out.println("\nExercise Check\n");  
        PrintStream outAlias = System.err;  
        Turtle tommy = new Turtle("Tommy", new Formatter(System.out)); //в System.out  
        Turtle terry = new Turtle("Terry", new Formatter(outAlias)); //в System.err  
        tommy.move(0, 0);  
        terry.move(4, 3);  
    }  
}
```

## Format Specifiers

- Format Синтаксис
  - `%[индекс $][флаги][ширина][точность]преобразование`
  - индекс это порядковый номер аргумента
  - флаги это «+» знак числа, «-» выравнивание по левому краю, по умолчанию по правому краю
  - ширина      число пробелов если не хватает, действует на `%s`, `%d`
  - точность      ширина `string`, `float`, `double`
  - преобразование это тип `%d,c,b,s,f,e,x,h,%` где `%%` это символ процента
- **ВНИМАНИЕ.** как обрезать строковый вывод до 15 символов `f.format("%-15.15s \n",name);`
- Пример
- ```
public class Receipt {  
    private double total = 0;  
    private Formatter f = new Formatter(System.out);  
    public void printTitle() {  
        f.format("%-25s %5s %10s\n", "Item", "Qty", "Price");  
        f.format("%-25s %5s %10s\n", "----", "----", "----");  
    }  
    public void print(String name, int qty, double price) {  
        f.format("%-15.15s %5s %10.2f\n", name, qty, price); //внимание как обрезать строку  
        total += price;  
    }  
    public void printTotal() {  
        f.format("%-15s %5s %10.2f\n", "Tax", "", total * 0.06);  
        f.format("%-15s %5s %10s\n", "", "", "----");  
        f.format("%-15s %5s %10.2f\n", "Total", "", total * 1.06);  
    }  
}
```

## StringFormat

- StringFormat
  - это аналог функции sprintf() который копирует форматированную строку в объект String
- Вывод файла в Hex
  - широко применяется String.format()
- Пример
 

```
public class DbEx extends Exception {
    public DbEx( int tId, int qId, String msg) { // messag version
        super(String.format("(t%d, q%d) %s",tId,qId,msg)); // String упакована внутри
                                                               // т.к. super() в 1й строке
    }
}
```

[lesson\\_ch12a/ch/ex6/local](#)

[lesson\\_ch12a/ch/ex6/include](#)

## RegEx регулярные выражения

- Регулярные выражения используются для групповой обработки

Символы		Символы	
.	любой символ	B	символ B
[abc]	любой из a,b,c	\xhh	символ в Hex
[^abc]	любой кроме a,b,c	\uhhhh	символ в Unicode
[a-zA-Z]	любой из диапазона a-Z	<b>Строки</b>	
[abc[hjk]	любой из a,b,c,h,j,k	\t	Tab
[a-z&&[hjk]]	любой из h,j,k пересечение	\n	LF
\s	пропуск (CR,LF,TAB,FF)	\r	CR
\S	НЕ пропуск [^\s]	\f	FF
\d	[0-9] любая цифра	\e	Esc
\D	НЕ цифра [^0-9]	<b>ВНИМАНИЕ.</b>	Для строки одна косая черта
\w	символ слова [a-zA-Z0-9]		
\W	символ НЕ слова [^\w]		
<b>Логические</b>		<b>Границы</b>	
XY	X за которым следует Y	^	начало строки
X Y	X или Y	\$	конец строки
(X)	группировка i-группа >> \i	\b	граница слова
		\B	НЕ граница слова
		\G	конец предыдущего совпадения

Max	Min	Catch		
X?	X??	X?+	X, один или ни одного	
X*	X*?	X*+	X, нуль и более	
X+	X+?	X++	X, один и более	
X{n}	X{n}?	X{n}+	X, ровно n раз повтор	
X{n,}	X{n,}?	X{n,}+	X, не менее n раз повтор	
X{n,m}	X{n,m}?	X{n,m}+	X, не менее n, но не более m раз	
<b>ВНИМАНИЕ.</b>	X+	abc+	значит ab, abc, abcc, abcccc	учитывать
		(abc)+	значит abc, abcabc, abcababc	группировку

Флаги	шаблонов		
	(?i)		case insensitive
	(?d)		unix line style
	(?x)		commentr ignore mode
	(?m)		multiline mode
	(?s)		dotall mode
	(?u)		case insensitive Unicode too

## Правила использования RegEx

- Правила использования регулярных выражений [lesson\\_ch12a/ch/ex7/local](#)
- -? правило X? может быть или нет знак «-»
- \\d правило \d любая цифра [\\ атрибут спец выражения Java](#)
- \\\\" просто \
- -?\d+ правило ?\d+ может – потом одна и более 0-9
- (-|\+)?\d+ может быть – или + потом одна и более 0-9
- \+\+ просто +
- Пример. проверка начало строки с заглавной буквы и конец строки точка [lesson\\_ch12a/ch/ex7/access](#) [lesson\\_ch12a/ch/ex9/access](#)

```
String s = "Look at the picture and fint the new objects.";
System.out.println(s.matches("(^[A-Z])(\\w+|\\W+)+([.]*$)"));
System.out.println(s.matches("(\\p{javaUpperCase}.*\\.)"));
// ^
// $                           это начало строки необязательно
// \\.                         это конец строки необязательно
// .*                          это ". [.]" тоже самое
// \p{javaUpperCase}           повтор любого символа сколько угодно раз
//                             универсальная конструкция для любого языка
System.out.println(s.replaceAll("[^aeiouAEIOU\\W]","_")); // Case sensitive
System.out.println(s.replaceAll("(?i)[aeiou\\W]","_")); // (?) Case insensitive
System.out.println(s.replaceAll("(?i)[aeiou]","_")); // Case insensitive
// (?) режим Case insensitive
```

## Создание RegEx

- создание RegEx полный список в `java.util.regex.Pattern` [lesson\\_ch12a/ch/ex10/local](#)
- Пример. все выражения regex соответствуют одной и той же строке

```
public class Local {
    public static void app() {
        String s = "Rudolph";
        String regex;
        regex = "(?i)[r][aeiou][a-z]ol.*";
        System.out.printf("s:%s regex: %25s match:%b\n",s,regex,s.matches(regex));
    }
}
```

## Quantifiers Квантификаторы

- Квантификатор описывает режим поглощения текста шаблоном
  - Greedy перебирает комбинации чтобы захватить максимальное число символов
  - Reluctant перебирает комбинации чтобы захватить минимальное число символов
  - Possessive супержадные, захватывают все что можно, и потом не уменьшают число символов чтобы добиться совпадения
- Пример [lesson\\_ch12a/ch/ex10/include](#)
- `public static void app() {`  
 `String s = "xoooooooooooofoo";`  
 `match(s,".*foo"); // greedy`  
 `match(s,".*?foo"); // relactance`  
 `match(s,".*+foo"); // possessive захватил всю строку по шаблону .* и на foo места нет`  
 `match(s,".{2}o*+foo"); // possessive удалось захватить только <oo..o> foo сработало`

## CharSequence

- CharSequence это интерфейс для последовательности символов

## Pattern и Matcher

- Pattern и Matcher классы для отработки вхождения pattern в строку

[lesson\\_ch12a/ch/ex10/include](#)

- Pattern это откомпилированная версия regex

[lesson\\_ch12a/ch/ex10/remote](#)

- Pattern.matches() метод для определения есть ли совпадение
  - Pattern.matcher() метод для генерации объекта Matcher

- Matcher это объект отвечающий за поиск regex в строке

- создается методом Pattern.matcher() при создании связывает regex и string
  - методы объекта find(), start(), end(), group()
  - ВНИМАНИЕ. обязательно вызывать find() на каждую итерацию поиска

- Пример.

```
public static void app() {
    String s = "abcabcaabcdefabc";
    String[] regexp = new String[] { "(abc+)", "(abc)+", "(abc){2,}" };
    System.out.println("String: "+s); // {2,} abc минимум 2 раза
    for (String r : regexp) {
        Pattern p = Pattern.compile(r); // скомпилировали pattern
        Matcher m = p.matcher(s); // связать pattern с выражением
        while (m.find()) {
            System.out.println("Match rx"+r+": \""+m.group()+""
                               "\\" at pos."+m.start()+"--"+(m.end()-1));
        }
    } // перебрать разные regexp
}
```

- Пример. крутой пример наборов regex

[lesson\\_ch12a/ch/ex10/access](#)

```
public static void app() {
    String s = "Java now has regular expressions";
    String[] regex = new String[]{"^Java", "\\Breq.*", "\\breg.*",
        "n.w\\s+h(a|i)s", "s?", "s*", "s+", "s{4}", "s{1}", "s{2}", "s{0,3}"};
    System.out.println("String: "+s);
    // ^Java в начале строки Java ok
    // \\Breq.* в середине слова reg и далее любые символы нет
    // \\breg.* в начале слова reg и далее любые символы ok
    // n.w\\s+h(a|i)s n.w любой символ \\s любой пробел h(a|i)s has | his ok
    // s? s или нет один раз ok
    // s* s или нет много раз ok
    // s{4} s ровно 4 раза нет
    // s{1} s ровно 1 раз ok
    // s{2} s ровно 2 раза ok
    // s{0,3} s от 0 до 3 раз ok
    for (String r : regex) {
        System.out.println("Regex: "+r);
        Pattern p = Pattern.compile(r);
        Matcher m = p.matcher(s);
        while (m.find()) {
            System.out.println("Match reg: "+r+" \""
                               "+m.group()+"\" at "
                               "+m.start()+" "+m.end());
        }
    }
}
```

- Пример. сложный regex

[lesson\\_ch12a/ch/ex11/access](#)

```
public static void app() {
    String s = "Arline ate eight apples and one orange while Anita hadn't any";
    String regex = "(?i)((^|[aeiou])|(\s+[aeiou]))\\w?[aeiou]\\b";
    // ^гласная + (1+) любой [гласная] на конце слова
    // пробел + гласная + (1+) любой [гласная] на конце слова
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while (m.find()) {
        System.out.println(m.group()+" "+m.start()+" "+m.end());
    }
}
```

## Matcher.find()

- find() функция поиска по строке по объекту CharSequence
  - работает как итератор проходит по всей строке
  - если задавать индекс, то поиск будет с того места какой индекс
  - **ВНИМАНИЕ.** если не менять индекс но задавать его легко попасть в бесконечный цикл
- Пример. [lesson\\_ch12a/ch/ex12/local](#)
- ```
public static void app() {
    Pattern p = Pattern.compile("\\w+"); // любые символы слов один и более
    Matcher m = p.matcher("Evening is full of the linnet's wings");
    while (m.find()) { // работает как итератор разбивает на слова
        System.out.print(m.group()+" . ");
    }
    System.out.println();
    int i = 0;
    while (m.find(i)) { // тоже самое, но поиск с заданной позиции
        System.out.print(m.group()+" . ");
        i++;
    }
}
```

## Matcher Groups

- Группы это части регулярного выражения заключенные в круглые () скобки
- A(B(C)D [ABCD] group0 [BC] group 1 [C] group 2
- Matcher методы работы с группами так [lesson\\_ch12a/ch/ex12/include](#)
  - groupCount() число групп кроме группы 0
  - group() выдать группу 0 после поиска find()
  - group(int) выдать найденную группу после поиска find()
  - start(int) индекс начала группы
  - end(int) индекс конца группы
- **ВНИМАНИЕ.** при работе с groupCount() использовать МЕНЬШЕ или РАВНО в цикле
- **ВНИМАНИЕ.** Matcher.find(0) включает все. Matcher.find(1) включает только [lesson\\_ch12a/ch/ex15/local](#) группу 1 можно использовать для отсечения маркеров, если вынести их за пределы группы
- Пример. [lesson\\_ch12a/ch/ex12/integra](#)

```
public static void app() {
    String poem = "Twas brillig, and the slithy toves\n" +
        "Did gyre and gimble in the wabe.\n" +
        "The frumious Bandersnatch.";
    String regex = "(?m) (\\\\S+)\\\\s+((\\\\S+)\\\\s+(\\\\S+))$";
    // (?m) multiline mode работает со строками внутри блока
    // \\S+ полезный символ много
    // \\s+ перевод строки много
    // структура строки (w) " ((w) " "(w) )$ три слова до конца строки
    // структура строки (w) " ((w) " "(w) )$ три слова до конца строки
    // распечатка групп 0: (w1) ((w2) (w3))
    // 1: (w1)
    // 2: ((w2) (w3))
    // 3: (w2)
    // 4: (w3)
    // 0.[the slithy toves]1.[the]2.[slithy]3.[toves]

    Matcher m = Pattern.compile(regex).matcher(poem);
    while (m.find()) {
        for (int i = 0; i <= m.groupCount(); i++) { // сколько групп захвачено
            System.out.print(i+"."+m.group(i)+"");
        }
        System.out.println();
    }
}
```

- Пример. вытащить все уникальные слова из текста

[lesson\\_ch12a/ch/ex12/access](#)

```
public class Access {
    public static void app() {
        String poem = "Twas brillig, and the slithy toves\n" +
                      "The frumious Bandersnatch.";
        String regex = "(\\b\\p{javaLowerCase}\\w+)" ; // групповое разбиение нужно для анализа
        // String regex = "\\b((?= [A-Z]) \\w+)" ; // (?= [A-Z]) по границам слов с прописной буквы
        // String regex = "\\b((?! [A-Z]) \\w+)" ; // (?! [A-Z]) по границам слов со строчной буквы
        Matcher m = Pattern.compile(regex).matcher(poem);
        Set<String> hset = new HashSet<>();
        while (m.find()) {
            hset.add(m.group());
        }
        Set<String> tset = new TreeSet<>(hset); // сразу сортировать
        System.out.println("Unic words:" + hset.size());
    }
}
```

- ВНИМАНИЕ.** конструкция отрицания и позитивного сравнения для группы `(?![A-Z])` `(?=[A-Z])`
- ВНИМАНИЕ.** отрицание для символа `[^a]` отрицание для группы `(?![A-Z])`

## Matcher start() и end()

- Matcher start() и end() функции адреса совпадения

[lesson\\_ch12a/ch/ex12a/local](#)

- заполняются только после успешного find() иначе вызов start() end() выдает IllegalStateException
- start() возвращает начальный индекс найденного совпадения
- end() возвращает индекс +1 конца совпадения

- Matcher matches() возвращает совпадение только с начала строки и только для всей строки

- Matcher lookingAt() возвращает совпадение только с начала строки но возможно только часть строки

- Пример.

[lesson\\_ch12a/ch/ex12a/access](#)

## Флаги шаблонов

- Флаги шаблонов

[lesson\\_ch12a/ch/ex14/local](#)

- ВНИМАНИЕ.** возможно подключение флагов прямо в Pattern конструкторе

```
Pattern p = Pattern.compile("^java", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
```

| Флаги шаблонов | Mode             | Режим поиска                                              |
|----------------|------------------|-----------------------------------------------------------|
| (?i)           | case insensitive | при поиске регистр игнорируется                           |
| (?x)           | comments         | игнорируются пробелы и текст после # до конца строки      |
| (?m)           | multiline mode   | маркеры ^ и \$ совпадают с началом и концом каждой строки |
| (?u)           | case insensitive | Unicode при поиске регистр игнорируется в режиме Unicode  |
| (?d)           | unix line style  | среди .<любой> ^ \$ распознается только \n                |
| (?s)           | dotall mode      | .<любой> включает и \n (по умолчанию не включает)         |

## String split()

- String split(regex)

разбивает строку по шаблону

[lesson\\_ch12a/ch/ex14/include](#)

- String split(regex,limit)

разбивает строку с ограничением размера выходного массива

- Пример.

[lesson\\_ch12a/ch/ex14/access](#)

- public static void app() {

```
String s = "This!!unusual use!!of exclamation!!points";
String[] a1 = Pattern.compile("!!").split(s); // простое разбиение Pattern
String[] a2 = Pattern.compile("!!").split(s,3); // ограничение длины Pattern
String[] a3 = s.split("!!"); // простое разбиение String
String[] a4 = s.split("!!",3); // ограничение длины String
System.out.println(Arrays.toString(a1)+" "+a1.length);
System.out.println(Arrays.toString(a2)+" "+a2.length);
}
```

## Replace operations

- Replace операции замены

- replaceFirst() замена первого совпадения regex
- replaceAll() замена всех вхождений regex
- appendReplacement() пошаговая замена разными regex
- appendTail() завершение после одного или нескольких appendReplacement()

- Пример. чтение блока текста из файла и форматирование

```
public class Local {  
    public static void app() {  
        /*! Here's a block of text to use as input to  
        the regular expression matcher. Note that we'll  
        first extract the block of text by looking for  
        the special delimiters, then process  
        the extracted block. !*/  
        String s = TextFile.read("src/ch/ex15/local/Local.java"); // читаем текущий файл  
        String regex = "/\\*!(.*)!\\*/"; // </ *! (любые символы много) !* /  
        Pattern p = Pattern.compile(regex, Pattern.DOTALL); // DOTALL \\. включает и \n  
        Matcher m = p.matcher(s);  
        if (m.find()) { // нашли строку  
            s = m.group(1); // выбираем только группу (.*) не содержит / *! и !* /  
            s= s.replaceAll(" {2,}", " "); // заменить 2 и более пробелов на 1  
            s=s.replaceAll("(?m)^ +","");
            System.out.println(s); // остались только одиночные пробелы  
            s = s.replaceFirst("[aeiou]", "(VOWEL1)");
            StringBuffer sb = new StringBuffer();
            Pattern p2 = Pattern.compile("[aeiou]"); // ищем все гласные
            Matcher m2 = p2.matcher(s);
            while (m2.find()) { // ищем все гласные строчные
                m2.appendReplacement(sb,m2.group().toUpperCase());
            }
            m2.appendTail(sb); // отрабатываем конец строки
            System.out.println(sb.toString());
        }
    }
}
```

[lesson\\_ch12a/ch/ex15/access](#)

## Matcher reset()

- Matcher reset() используется для работы объекта matcher с несколькими строками
  - reset() сбрасывает объект на начало текущей последовательности
  - reset(str) переключает объект на новую строку с текстом
- Пример.

[lesson\\_ch12a/ch/ex15/include](#)

## Regex и InOut

- Regex и InOut
    - regex очень широко используются в поиске и операциях ввода, вывода
    - grep наиболее распространенная программа поиска в файлах
- ```
public class JGrep {  
    private String fName;  
    private String regex;  
    public JGrep(String fName, String regex) {  
        this.fName = fName;  
        this.regex = regex;  
    }  
    public void find() {  
        ArrayList<String> alist;  
        if (fName.length() == 0 || regex.length() == 0) {  
            System.out.println("Usage: fileName, regex");  
            return;  
        }  
        try {  
            alist = new TextFile(fName);  
        } catch (Exception e) {  
            System.out.println("File Not Found: "+fName);  
            return;  
        }  
        Pattern p = Pattern.compile(regex);  
        Matcher m = p.matcher(""); // подготовили объект для работы со строками  
        int index = 1; // перебор строк  
        for (String s : alist) {  
            m.reset(s); // подключили строку  
            while (m.find()) {  
                System.out.printf("%2d: %-10s %2d\n", index++, m.group(), m.start());  
            } // показывает позицию в строке  
        }  
    }  
}
```

[lesson\\_ch12a/ch/ex15/integra](#)

## Folder InOut

- Folder InOut    работа с каталогами
  - отличный класс работы с файлами File[] file, методы listFiles(), isDirectory(), getPath().
- ```
private static final int L_MAX = 2;  
private static int level = 0;  
private static void fp(File file, String regex) {  
    JGrep jp = new JGrep(file.getPath(), regex, "NOCA");  
    jp.find();  
}  
public static void fd(File file, String regex) { // рекурсивная обработка каталогов  
    File[] files;  
    level++; // на следующий уровень  
    if (file.isDirectory()) { // это директорий  
        files = file.listFiles();  
    } else {  
        files = new File[]{file};  
    }  
    for (File f : files) {  
        if (f.isDirectory()) {  
            if (level < L_MAX) {  
                fd(f, regex); // рекурсивный вызов  
            } else {  
                System.out.printf("EXIT LOOP: %s\n", f.getParent());  
            }  
        } else {  
            fp(f, regex);  
        }  
    }  
    level--; // вышли
```

[lesson\\_ch12a/ch/ex16/access](#)

- Пример. чтение комментариев

[lesson\\_ch12a/ch/ex17/access](#)

- ключевой это regex в котором надо соблюсти 3 условия
- первое: прочитать файл вместе с «\n» с помощью TextFile.read()
- второе: в односторочных взять максимально жадный (.\*)
- третье: в многострочных взять минимально жадный (.\*)?
- четвертое: в многострочных прихватить \n в качестве .\* (?s) режим захвата \n

- ВНИМАНИЕ. ниже в regex2 показано как разместить комментарии внутри RegEx возможно благодаря (?x)

```
public static void app() { //1 еще один комментарий
    // String regex = "(?m) (?s) (?x) (//(.*?))|/\*\*(.*?)\*/";
    // (?m) многострочный обязательно
    // (?s) захват \n конструкцией .* для многострочников
    // (?x) игнорирует все то после # до конца строки показано в regex2
    // (//(.*?)) группа // *** \n по минимуму
    // "/\*\*(.*?)\*/" #захват многострочников /*...*/ по минимуму\n";
    String fName = "src/ch/ex17/access/Access.java";
    String s = TextFile.read(fName); // условие чтение в файл
    String regex = "(?m) (?s) (?x) (//(.*?))|/\*\*(.*?)\*/";
    String regex2 =
        "# многострочный режим \n" +
        "# первое условие захват \n" +
        "# режим комментариев внутри regex \n" +
        "# захват участков // ***$ по минимуму \n" +
        "|" +
        "/\*\*(.*?)\*/" #захват многострочников /*...*/ по минимуму\n";
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while (m.find()) {
        System.out.println(m.group());
    }
    JGrepFile.showDemo(fName, regex2);
}
```

- Пример. Выборка имени класса

[lesson\\_ch12a/ch/ex19/access](#)

- имя класса это последовательность символов
- имя расширяемого или конструктора класса это слова и возможно через точку
- 

- public static void app() {**

```
String sClass = "[A-Za-z_][A-Za-z0-9]*"; // имя класса Abbb a999
String sInter = sClass +"(\\." +sClass+ ")*"; // повторяется группа abbb .abbb .bccc
String regex = "class\\s"+sClass+"|"+ // class Abbb
              "extends\\s"+sInter+"|"+ // extends Abbb.Bccc.Cddd
              "new\\s"+sInter; // new Abbb.Bccc.Cddd
String fName = "src/ch/ex19/access/Access.java"; // первое условие чтение в файл
String s = TextFile.read(fName);
Internal.Enclosed internal = new Internal.Enclosed();
Internal.Enclosed.Embossed embossed = new Internal.Enclosed.Embossed();
Iterator<String> it = new Iterator<String>() {
    public boolean hasNext() {
        public String next() {
    };
    String regex2 = sInter;
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while (m.find()) {
        System.out.println(m.group().replaceFirst("new\\s|class\\s|extends\\s", ""));
    }
}
```

## Сканирование ввода

- BufferedReader класс потокового ввода вывода из буфера
  - позволяет выбирать из потока данные построчно
  - использует StringTokenizer для инициализации
- StringTokenizer преобразует объект String в поток строк

[lesson\\_ch12a/ch/ex20/include](#)

## Scanner

- Scanner универсальный класс потокового ввода, вывода
  - работает с потоками, файлами, консолью
  - содержит метод next() для всех примитивных типов
  - поглощает IOException , которые доступны в методе IOException()

[lesson\\_ch12a/ch/ex20/access](#)

## Scanner RegEx Delimiters

- Scanner имеет свои собственные ограничители строки
  - разделители в строке можно переназначить scanner.useDelimeter(regex)
- Пример
- ```
public static void app() {
    Scanner scanner = new Scanner("12 , 42, 53, 89 , 46# 09. 12");
    scanner.useDelimeter("\s*[.,#]\s*"); // новая группа разделителей , или . или #
    while (scanner.hasNext()) {
        System.out.println(scanner.next());
    }
}
```

[lesson\\_ch12a/ch/ex20/integra](#)

## Scanner RegEx Scan

- Scanner и регулярные выражения при сканировании
  - regex позволяет по шаблону выискивать в потоке текста нужные строки
  -
- **ВНИМАНИЕ.** в функции hasNext() тоже должен быть шаблон иначе выдаст исключение
- Пример. поиск и выдача адресов интернета и даты связи

```
String s =
    "58.27.82.161@02/10/2005\n" +
    "204.45.234.40@02/11/2005\n" +
    "58.27.82.161@02/11/2005\n" +
    "58.27.82.161@02/12/2005\n" +
    "58.27.82.161@02/12/2005\n" +
    "[Next log section with different data format]";

Scanner scanner = new Scanner(s);
String regex = "(\d+\.\d+\.\d+\.\d+)\@(\d{2}/\d{2}/\d{4})"; // pattern
while (scanner.hasNext(regex)) {
    scanner.next(regex);
    MatchResult matchResult = scanner.match(); // найти результат
    String ip = matchResult.group(1);
    String date = matchResult.group(2);
    System.out.printf("Link on %s from ip %s\n", date, ip );
}
```

[lesson\\_ch12a/ch/ex20/remote](#)

## String Tokenizer

- устаревший класс для работы с Scanner Regex
- **ВНИМАНИЕ.** StringTokenizer работает с шаблонами сложнее чем Scanner
- Пример.

[lesson\\_ch12a/ch/ex20/remote](#)

## RTTI

- RTTI динамическое определение типов
  - при восходящем преобразовании типы потомков «стираются» при помещении в массив предка
  - при извлечении объекта из массива происходит динамическое определение типа потомка
  - это и называется RTTI
  - далее на базе RTTI начинает работать полиморфизм

[lesson\\_ch14/ch/ex1/local](#)

## Объект Class

- Объект Class специальный объект который используется при создании объекта любого класса
- Последовательность загрузки программы
  - загрузчик проверяет загружен ли объект Class для данного типа
  - далее поиск файла \*.class, загружается байт код с проверкой, объект Class загружается в память
  - затем объект Class данного типа используется для создания всех объектов данного типа
  - по сути это ссылка на тот самый статический Class от которого работают статические методы
- Методы и поля объекта Class
  - `Class.forName()`                  метод вызывает объект Class
  - `Class.class`                  поле дает ссылку на объект Class
  - `getName()`                  package.имя класса
  - `getSimpleName()`                  имя класса
  - `getCanonicalName()`                  package.имя класса
  - `newInstance()`                  создает объект класса но только Object
  - `final static`                  если нет активной инициализации
  - если есть например `Random().getInt()`

Создание объекта Class

создает

не создает

- Пример. инициализация создание объекта методом объекта Class

[lesson\\_ch14/ch/ex1/access](#)

```
public static void app() {  
    Class c = null; // объект типа Class  
    try {  
        c = Class.forName("ch.ex1.access.FancyToy"); // получить ссылку на класс  
                                                 // ВНИМАНИЕ указан путь к классу  
    } catch (Exception e) {  
    }  
    Class up = c.getSuperclass(); // ссылка на объект Class предка Toy  
    Object o = null;  
    try {  
        o = up.newInstance(); // нужен конструктор Toy по умолчанию  
    } catch (InstantiationException e) {  
    } catch (IllegalAccessException e) {  
    }  
    printInfo(o.getClass()); // получить объект Class из объекта  
}
```

- Пример. cast восходящее нисходящее

[lesson\\_ch14/ch/ex3/access](#)

```
public static void app() {  
    Shape shape = new Rhomb(); // upcast Rhomb to Shape  
    Rhomb rhomb = (Rhomb)shape; // downcast Shape(Rhomb) to Rhomb  
    Circle circle = (Circle)shape; // downcast Shape(Rhomb) to Circle  
    rhomb.draw();  
    circle.draw();  
}
```

// выдаст ошибку только при Runtime

## Доступ к объекту Class

- Доступ к объекту либо создает статический объект Class либо нет, в зависимости от типа доступа
  - Class загружается при доступе к методам или полям которые инициализируются методами
  - Class НЕ загружается при доступе к \*.class или полям константам которые заданы НЕ методами
- Доступ к объекту Class возможен прямо через имя класса ClassName.class
  - доступ к ссылке на объект Class ClassName.class не загружает объект Class
  - ссылка на Class.class дает возможность создавать объекты данного класса
  - объект Class можно найти двумя путями через поле \*.class или через метод forName()

- Пример. инициализация классов с командной строки

[lesson\\_ch14/ch/ex7/access](#)

- создается экземпляр класса через имя класса

```
public class Access {  
    private static final String[] C_NAMES = {"Candy", "Cookie", "Gum"};  
    public static void app(String[] args) {  
        if (args == null || args.length == 0) {  
            System.out.println("Usage:java ch.ex7.Ex7 Candy or Cookie,Gum");  
            return;  
        }  
        Class c = null;  
        String pkg = Candy.class.getPackage().toString().replaceFirst("p\\\\w+\\\\s", "");  
        for (String arg : args) {  
            for (int i = 0; i < C_NAMES.length; i++) {  
                String regex = "(?i) (" + C_NAMES[i].substring(0, 2).toLowerCase() + "\\w+");  
                if (arg.matches(regex)) {  
                    try {  
                        c = Class.forName(pkg + ". " + C_NAMES[i]);  
                        Object cc = c.newInstance();  
                        System.out.println("created: " + c.getSimpleName());  
                    } catch (Exception e) {  
                        System.out.println("Can't create Instance");  
                    }  
                }  
            }  
        }  
    }  
}
```

## Литералы .class

- Литералы .class это официальная ссылка на объект класса в JVM
  - например Ball.class – это ссылка на Объект класса Ball
  - все примитивные типы имеют литералы класса .class и .TYPE оба ссылаются на тот же объект
- **ВНИМАНИЕ.** Обращение к литералу class или TYPE НЕ загружает объект класса
- 

	Object		Object
boolean.class	Boolean.TYPE	long.class	Long.TYPE
char.class	Character.TYPE	float.class	Float.TYPE
byte.class	Byte.TYPE	double.class	Double.TYPE
short.class	Short.TYPE	void.class	Void.TYPE

- 
- Процедура загрузки объекта Class
  - Загрузка. Загрузчик находит байт-код и создает объект Class
  - Компоновка. Проверяется байт-код, память для статических полей, ссылки на другие классы
  - Инициализация. Выполняется инициализация суперкласса если надо, статических полей и блоков
- **ВНИМАНИЕ.** Инициализация откладывается до первого вызова статическому метода или non-final поля
- Пример.

[lesson\\_ch14/ch/ex11/local](#)

## Class ссылка на обобщенные классы

- Class это ссылка на обобщенный класс, как Object работает с любым классом [lesson\\_ch14/ch/ex11/integra](#)
  - **ВНИМАНИЕ.** Вместо Class n; использовать Class<?> n; это предпочтительный вариант
- Class<? extends T> это ссылка на обобщенный класс ограниченный классом T и его subclasses
  - как вариант Class<? extends Number> все подклассы абстрактного класса Number
  - посмотреть подклассы можно в документации, проверить getInfo(Integer.class)
- Пример [lesson\\_ch14/ch/ex11/include](#)

```
public static void app() {  
    Class intClass = int.class;  
    intClass = double.class; // сработало  
    Class<Integer> integerClass = int.class; // сработало  
    integerClass = Integer.class; // сработало  
    //integerClass = double.class; // не работает Double не расширяет Integer  
    // Class<Number> numNumClass = int.class; // странно но так НЕ работает  
    Class<?> numClass = int.class; // работает  
    numClass = double.class; // работает  
    Access.getInfo(Byte.class);  
    Access.getInfo(Integer.class);  
    Access.getInfo(Double.class);  
    Class<? extends Number> numExtClass = int.class; // так работает для всех  
    numExtClass = Integer.class; //subclasses of Number  
    numExtClass = Float.class;  
    numExtClass = float.class;  
    numExtClass = double.class;  
    numClass = Number.class;  
}
```

## Динамическое создание экземпляров через объект класса

- Объект Class позволяет создавать экземпляры класса
  - при этом используется метод Class.getInstance()
- **ВНИМАНИЕ.** Есть ограничения на объекты созданные таким образом
- Есть ограничения на объекты созданные таким образом
  - пускай есть объект <FancyToy extends Toy>
  - FancyToy.class.getSuperClass() **НЕЛЬЗЯ** создать объект Class< Toy>  
◦ **МОЖНО** создать объект Class <? super FancyToy>
  - пускай получен объект ftSuper типа <? super ClassName>
  - методом ftSuper.newInstance() **НЕЛЬЗЯ** создать объект Toy obj Compiler  
◦ **НЕЛЬЗЯ** создать объект Object obj Runtime
  -

- Пример.

```
public static void app() throws Exception {  
    Class<FancyToy> ftClass = FancyToy.class; // ссылка на объект Class  
    FancyToy f = ftClass.newInstance(); // экземпляр создает  
    Toy t = ftClass.newInstance(); // экземпляр восходящее  
    // Class<Toy> tt2 = ft.getSuperclass(); // так не работает  
    // Toy t4 = tt2.newInstance(); // не работает  
    Class<? super FancyToy> ftSuper = ftClass.getSuperclass(); // работает  
    Object t4 = tt2.newInstance(); // НЕ работает RunTime  
    Class<Toy> tSuper = Toy.class; // работает  
    Toy t2 = tt.newInstance(); // НЕ работает RunTime  
    // Cast  
    System.out.println("f :" + f.getClass().getSimpleName());  
    System.out.println("t :" + t.getClass().getSimpleName());  
    System.out.println("tt2:" + ftSuper.getSimpleName());  
    System.out.println("tt :" + tSuper.getSimpleName());  
}
```

## Class приведение типа

- Class приведение типа есть два вида синтаксиса
  - object = typeClass.cast(object2)                    новый вариант
  - object = (Class)object2                            стандартный вариант
- Пример.

```
public static void app() throws Exception {
    Class<FancyToy> ftClass = FancyToy.class; // ссылка на объект Class
    Toy t = ftClass.newInstance(); // экземпляр восходящее тоже создает
// Cast
    FancyToy h2 = ftClass.cast(t); // обратный cast
    FancyToy h4 = (FancyToy)t; // обратный cast
    h2.draw();
    h4.draw();
}
```

## Проверка перед приведением типов

- Способы динамического приведения типов RTTI
  - стандартное прямое приведение типа (Shape)object
  - запрос информации у object.getClass() или напрямую у объекта Class
  - конструкция if( obj instanceof Class ) проверки объекта на принадлежность к классу
- Применение instanceof
  - ВСЕГДА проверять принадлежность объекта прежде принудительного приведения типа
- **ВНИМАНИЕ.** Если в программе нагромождение instanceof надо пересмотреть архитектуру программы
- 
- Пример. Два способа идентификации типа
- Вариант 1. Определение типа класса getClass()==Class.class

```
public void count(Pet pet, List<Class<? extends Pet>> types) {
    for (Class<? extends Pet> typeClass : types) {
        if (pet.getClass() == typeClass) {
            count(pet.getClass().getSimpleName());
        }
    }
}
```

- Вариант 2. Определение типа класса на этапе компиляции instanceof

```
public void count(Pet pet) {
    if (pet instanceof Pet) {
        count("Pet");
    }
    if (pet instanceof Dog) {
        count("Dog");
    }
    if (pet instanceof Cat) {
        count("Cat");
    }
    if (pet instanceof Rodent) {
        count("Rodent");
    }
}
```

## Использование литералов class

- Использование литералов class
  - это обращение к объекту класса из экземпляра или названия класса
  - дает статический доступ к объекту класса без загрузки объекта класса
- Пример. очень мощный пример создания списка

[lesson\\_ch14/ch/ex11/thrown](#)

## Динамическая проверка типа

- Динамическая проверка типа Class

Приложение D

[lesson\\_ch14/ch/ex11/remote](#)

```
Class<FancyToy> ftClass = FancyToy.class; //объект Class
Toy t = ftClass.newInstance();
FancyToy h2 = ftClass.cast(t);           // обратный cast равносильно FancyToy(t)
<FancyToy.class.isInstance(h2)>
<FancyToy.class == h2.getClass()>
<Toy.class == h2.getClass()>           // не компилируется
<FancyToy.class == h2.getClass().getSuperclass()>
<FancyToy.class.isAssignableFrom(h2.getClass())>
```

- Пример. проверка < ClassA == ClassB >

[lesson\\_ch14/ch/ex11/value](#)

```
public void count(Pet pet, List<Class<? extends Pet>> types) {
    for (Class<? extends Pet> typeClass : types) {
        if (pet.getClass() == typeClass)
            count(pet.getClass().getSimpleName());
    }
}
```

- Пример. проверка < ClassA.isInstance(ClassB) >

```
private void count(Pet pet) {           // так класс наследует
    for (Map.Entry<Class<? extends Pet>, Integer> entry : entrySet()) {
        if (entry.getKey().isInstance(pet))
            put(entry.getKey(), entry.getValue() + 1);
    }
}
```

- Пример. проверка < ClassA.isAssignableFrom(ClassB) >

```
public void count(Object object) {
    Class<?> type = object.getClass();           // получить класс от входного объекта
    if (!baseType.isAssignableFrom(type)) {         // сравнение <baseType> n = new <type>
        throw new RuntimeException("catch: wrong type :" + type.getSimpleName() +
            " must be subclass of:" + baseType.getSimpleName());
    }
    countClass(type); // посчитать
}
```

## Итераторы и Class

- Итераторы варианты реализации

[lesson\\_ch14/ch/ex12/access](#)

- Пример. стандартный тип класс возвращает свои экземпляры

```
public class Ball implements Iterable<Ball> {
    @Override
    public Iterator<Ball> iterator() {
        return new Iterator<Ball>() {
            public boolean hasNext() {
                return null;
            }
            @Override
            public Ball next() {
                return null;
            }
        };
    }
    public class BallInt implements Iterable<Integer> {
        @Override
        public Iterator<Integer> iterator() {
            return new Iterator<Integer>() {
                public boolean hasNext() {
                    return null;
                }
                @Override
                public Integer next() {
                    return null;
                }
            };
        }
    }
}
```

## Registered Factory

- Registered Factory это механизм генерации объектов классов автоматом
  - Есть два способа это подкласс генератора объекта или генератор на основе объекта Class
  - Фабрика подкласс когда класс содержит подкласс с интерфейсом генератора экземпляра класса
  - Генератор объекта можно создавать из экзегeneric <T> ??? непонятно зачем нужно??

## Factory Generic Interface<T>

- Стандартная реализация

```
private static List<IFactory> pList = new ArrayList<>(); // IFactory это интерфейс
```
- Расширенная реализация

```
private static List<IFactory<?extends Part>> pList = new ArrayList<>(); // IFactory<T>
```
- ВНИМАНИЕ. List<IFactory<?extends Part>> pList означает Class или Interface с Generic <T>  
IFactory<T> где <T> == <?extends Part> Поэтому IFactory<T> == <IFactory<?extends Part>>
- Пример. Стандартная реализация  
[lesson\\_ch14/ch/ex14/access](#)
- ```
public interface IFactory {  
    Part create();  
}
```
- ```
public class Part {  
    private static List<IFactory> pList = new ArrayList<>(); // IFactory это интерфейс  
    static {  
        // static block  
        pList.add(new FuelFilter.Factory()); // список внутренних классов генераторов  
        pList.add(new AirFilter.Factory()); // все работают по интерфейсу IFactory  
    }  
    public static Part createRandom() {  
        int n = rnd.nextInt(pList.size()); // случайным образом детальку из списка  
        return (Part) pList.get(n).create(); // создается из типа деталька экземпляр  
    }  
}
```
- ```
public class FuelFilter extends Filter implements IFactory {  
    public static class Factory implements IFactory { // статический подкласс генератор  
        @Override  
        public FuelFilter create() {  
            return new FuelFilter();  
        }  
    }  
}
```
- Пример. Расширенная реализация  
[lesson\\_ch14/ch/ex12a/access](#)
- ```
public interface IFactory<T> {  
    T create();  
}  
public class FuelFilter extends Filter {  
    public static class Factory implements IFactory<FuelFilter> { // класс генератор  
        @Override  
        public FuelFilter create() {  
            return new FuelFilter();  
        }  
    }  
}  
public class Part {  
    private static List<IFactory<?extends Part>> pList = new ArrayList<>(); // IFactory<T>  
    static { // static block  
        pList.add(new FuelFilter.Factory()); // список статических экземпляров генераторов  
        pList.add(new AirFilter.Factory());  
    }  
    public static Part createRandom() {  
        int n = rnd.nextInt(pList.size()); // случайным образом детальку из списка  
        return (Part) pList.get(n).create(); // создается из типа деталька экземпляр  
    }  
}
```

- Фабрика Class когда экземпляр класса генерируется при помощи метода Class.newInstance()
- Пример
- ```
public class Part {
    private static List<Class<?extends Part>> pList = new ArrayList<>(); //
    private static Random rnd = new Random();
    static { // static block
        pList.add(FuelFilter.class); // это пока только список типов деталек пустой
        pList.add(AirFilter.class);
    }
    public static Part createRandom() { // точно также создает объекты Part
        int n = rnd.nextInt(pList.size()); // случайным образом детальку из списка
        try {
            if (!Part.class.isAssignableFrom(pList.get(n))) {
                System.out.println("catch: Class does not extend Part");
                throw new RuntimeException();
            }
            return (Part) pList.get(n).newInstance(); // создается из типа экземпляра
        } catch (Exception e) {
            System.out.println("catch: Class is not compatible"+e.getMessage());
            throw new RuntimeException();
        }
    }
    @Override
    public String toString() {
        return getClass().getSimpleName();
    }
}
```

## InstanceOf и сравнение объектов Class

- Есть несколько способов определить принадлежность объекта к классу
- obj instanceof Class определить принадлежность объекта к классу и предкам
- Class.isInstance(obj) тоже самое
- obj.getClass() == Class.class точное равенство к конкретному классу предков не берет
- obj.getClass().equals(Base.class) точное равенство к конкретному классу предков не берет
- Пример.
- System.out.println("Test x type : "+x.getClass());
 System.out.println("x instanceof Base : "+(x instanceof Base));
 System.out.println("Base.isInstance(x) : "+Base.class.isInstance(x));
 System.out.println("x.getClass()==Base.class : "+(x.getClass() == Base.class));
 System.out.println("x.getClass().equals(Base.class) : "+x.getClass().equals(Base.class));

## Reflections Динамическая информация о классе

- Отражение Динамическая информация о классе
- если Class объекта неизвестен его можно получить от функциями работы с объектом Class
- ВНИМАНИЕ.** Тип объекта можно получить только если он известен на этапе компиляции
- Reflections это набор методов получения информации о неизвестном классе из экземпляра
- getFields() получить поля объекта
- getMethods() получить методы объекта
- getConstructors() получить конструкторы объекта
- Извлечение информации о методах класса
- используя возможности Reflection можно вытащить методы класса и его предков
-

## Reflection применение

- Reflection применение позволяет создавать объекты вытаскивая конструктор из объекта
  - можно вытащить конструктор конкретно объекта
  - можно вытащить конструктор класса предка
- Пример. вытащить конструктор с параметрами и создать объект
- Пример. мощный пример рекурсивной обработки данных класса

[lesson\\_ch14/ch/ex19/access](#)

[lesson\\_ch14/ch/ex20/access](#)

```
private static Object getSuperObject(String cName, String cVal) {  
    //ToyTest  
    Class<?> c = null; // объект типа Class  
    try {  
        c = Class.forName(cName); // получить ссылку на класс  
    } catch (Exception e) {  
        System.out.println("Class not found");  
        throw new RuntimeException();  
    }  
    Constructor<?>[] cons = c.getSuperclass().getConstructors();  
    Object s = null; // заготовка для объекта  
    for (Constructor con : cons) {  
        Class[] params = con.getParameterTypes();  
  
        if (params.length > 0) {  
            try {  
                if (params[0] == int.class) {  
                    s = con.newInstance(Integer.parseInt(cVal)); // стандартный  
                    return s;  
                }  
                if (params[0] == Integer.class) {  
                    s = con.newInstance(new Object[]{ Integer.valueOf(cVal) });  
                    return s; // более универсальный  
                }  
            } catch (Exception e) {  
                System.out.println("Constructor failed:" + e.getMessage());  
                throw new RuntimeException();  
            }  
        }  
    }  
    System.out.println("Construction failed.");  
    return null; // не удалось создать объект  
  
}  
  
public static void app() {  
    //ToyTest  
    Object s = getSuperObject("ch.ex19.access.toys.SuperToy", "120");  
    System.out.println(s.getClass().getSimpleName() + ":" + s);  
    System.out.println(s instanceof ExtToy);  
    ExtToy.class.cast(s).show();  
    ExtToy.class.cast(s).list();  
  
    ExtToy s2 = (ExtToy) getSuperObject("ch.ex19.access.toys.SuperToy", "120");  
    System.out.println(s2.getClass().getSimpleName() + ":" + s2);  
    s2.list();  
    s2.show();  
}
```

## Dynamic Proxy

- Standard Proxy это один из паттернов проектирования
  - создается интерфейс A
  - класс объекта реализует интерфейс A
  - класс proxy тоже реализует интерфейс A и содержит в себе объект типа интерфейс A
  - методы proxy это тоже интерфейса A, выполняют работу proxy и вызывают методы объекта

[lesson\\_ch14/ch/ex21/local](#)

- Пример.

```
public interface IFace {  
    void doSome();  
    void someElse(String s);  
}  
  
public class SimpleProxy implements IFace {  
    private IFace proxioObject; // ссылка на объект с интерфейсом IFace  
    public SimpleProxy(IFace proxioObject) {  
        this.proxioObject = proxioObject;  
    }  
    public void doSome() {  
        System.out.print("SimpleProxy.doSome > "); // функции Proxy  
        proxioObject.doSome(); // функции Real за Proxy  
    }  
    public void someElse(String s) {  
        System.out.print("SimpleProxy.someElse."+s+" > "); // функции Proxy  
        proxioObject.someElse(s); // функции Real за Proxy  
    }  
}
```

- Dynamic Proxy Концепция

- в обычном Proxy объект Proxy содержит столько же методов сколько и proxied объект
- так как процедура Proxy обычно едина для всех, смысла дублировать все методы нет,
- Dynamic Proxy предлагает метод invoke() , который вызывается для всех методов proxied объекта

- Общая последовательность работы такая

- proxy.invoke выполняет промежуточные операции, аргументы invoke() объект, метод и аргументы
- затем через invoke(объект, аргументы) оригинальный метод объекта proxied

- **ВНИМАНИЕ.** Proxy.toString() внутри invoke() даст stackOverflow, так как toString тоже метод и Proxy

пытается заместить его как все методы и вызывает invoke() внутри которого вызывается ... toString()

- Пример.

[lesson\\_ch14/ch/ex23/access](#)

```
public class DmProxy implements InvocationHandler {  
    private Object proxyObj;  
    public DmProxy(Object proxyObj) {  
        this.proxyObj = proxyObj;  
    }  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        System.out.println("Proxy:");  
        System.out.println(loopcount+" "+super.toString()); // отработает нормально  
        System.out.println(proxy.toString()); // вызывает stackOverflow  
        return method.invoke(proxyObj, args); // выполнить оригинальный метод  
    }  
}  
  
public static void app() {  
    IFace r = new RealObject();  
    IFace proxy = (IFace)Proxy.newProxyInstance(  
        IFace.class.getClassLoader(),  
        new Class[]{IFace.class},  
        new DmProxy(r)  
    );  
    r.doSome(); // выполнить оригинальный метод  
    r.someElse("data"); // выполнить оригинальный метод  
    proxy.doSome(); // выполнить через прокси  
    proxy.someElse("data"); // выполнить через прокси  
}
```

## Dynamic Proxy

- Dynamic Proxy
  - Proxy Class это класс которые создает объект proxy динамически
  - Proxy.newProxyInstance() метод создания proxy, в аргументах объект помещаемый за proxy
  - Interface InvocationHandler интерфейс для класса ProxyHandler, который отвечает за замещение
- Процедура создания proxy объекта
  - все довольно формально и реализуется в 2 этапа
  - создать класс в котором реализовать InvocationHandler interface
  - вызвать статический метод newProxyInstance()
- InvocationHandler
  - получает аргументы <Object экземпляр объекта>, <Method метод>, <аргументы Object[] args>
  - метод запускается стандартной процедурой класса Method.invoke(Object, args)
  - берем объект, указываем ему какой метод запустить и отдаём аргументы
- Реализация Dynamic Proxy
- Пример.

[lesson\\_ch14/ch/ex21/include](#)

```
public class DProxy implements InvocationHandler{  
    private Object proxiObj;  
    public DProxy(Object proxiObj) {  
        this.proxiObj = proxiObj;  
    }  
    public Object invoke(Object proxy, Method met, Object[] args) throws Throwable{  
        // ... промежуточные функции Proxy  
        return met.invoke(proxiObj, args);  
    }  
}
```

## Null объекты

- Null объекты
  - встроенный объект null не имеет методов и может выдать только NullPointerException
  - чтобы эффективно использовать null в отношении классов используют заменители «Null Object»
- «Null-объекты» это объект класса пользователя с особыми свойствами
  - объект имеет свойства аналогичные null, при этом имеет все поля и методы класса пользователя
  - «Null Object» это паттерн проектирования частный случай паттерна «Стратегия»
- Пример. создание объекта «Null Object» для класса Person
  - поля объекта имеют значение String("none") что соответствует состоянию «null»

[lesson\\_ch14/ch/ex24/local](#)

```
public class Person {  
    public final String firstName;  
    public final String lastName;  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    public static class NullPerson extends Person implements Null{ // встроенный класс  
        private NullPerson() {  
            super("None", "None");  
        }  
        @Override  
        public String toString() {  
            return "NullPerson";  
        }  
    } // класс NullPerson  
    public static final Person NULL = new NullPerson(); // экземпляр класс SingleTone  
    @Override  
    public String toString() {  
        return "Person:" + firstName + " " + lastName + " " + address;  
    }  
} // класс Person
```

## toString() параметры объекта

- `toString()` параметры объекта выводится имя объекта @`hashCode().HEX`  
`return this.getClass().getName()+"@"+Integer.toHexString(this.hashCode());`

## Dynamic Proxy for Null Object

- Dynamic Proxy for Null Object lesson\_ch14/ch/ex24/include
- Процедура состоит из двух этапов
  - Создание робота `SnowRobot` в интерфейс встроен встроенный класс для тестирования
  - Создание динамического Proxy довольно сложный генератор
- Пример генератора Dynamic Proxy for NullObject
- ```
public class NullRobotProxy implements InvocationHandler {  
    private String nullName; // имя для пустого объекта робот  
    private Robot proxyObj = new NRobot(); // встроенный класс  
  
    private class NRobot implements Null, Robot { // оба интерфейса проприетарные  
        public String name() {  
            return nullName;  
        }  
        public String model() {  
            return nullName;  
        }  
        public List<Operate> operate() {  
            return Collections.emptyList(); // возвращает пустой список  
        }  
    } // class NRobot finished
```
- ```
// конструктор
```
- ```
public NullRobotProxy(Class<? extends Robot> type) { // производные от Robot  
    nullName = type.getSimpleName()+"NULLRobot"; //NULL имени класса объекта  
}
```
- ```
// метод InvocationHandler  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        return method.invoke(proxy, args); // вызвать оригиналый метод  
    }
```
- Генератор объекта NullObject
- **ВНИМАНИЕ.** два класса в массиве классов, это классы интерфейсов встроенного класса NRobot
- ```
public static Robot getNULL(Class<? extends Robot> type) { // единственный метод  
    создания объекта  
    return (Robot) Proxy.newProxyInstance(  
        NullRobot.class.getClassLoader(),  
        new Class[] { Null.class, Robot.class },  
        new NullRobotProxy(type)  
    );  
}
```

## Стандартное применение Null

- Стандартное решение Null внутренний класса и объект static lesson\_ch14/ch/ex24/access
- Встраивание в виде класса нужно чтобы
  - создать объект который отрабатывает все методы интерфейса или класса особым образом
  - создать объект который отрабатывает `toString()` особым образом
  - подключить интерфейс `INull` чтобы объект откликался на этот тип класса
- Статический объект
  - объект Null должен быть статическим, чтобы он был один на все объекты данного класса
- Уровень класса с подклассом Null
  - встраивать надо на уровне минимальной детализации
  - если нужна детализация `Part >> (Filter, Belt)` то на уровне `Filter.NULL` и `Belt.NULL`
  - если нужна детализация `Part` то на уровне `Part.NULL`
- Доступ к объекту Null
  - стандартный доступ через имя класса, как к статическому объекту
  - как вариант сделать доступ через метод, прототип которого в корневом предке
- ```
public interface IPart {
    String getModel();
    Part getNull();
}

abstract public class Part implements IPart {
    // показано только то, что касается Proxy в данном классе это толькo интерфейс IPart
}

public class AirFilter extends Filter {
    public static AirFilter NULL = new NullAirFilter();

    public static class Factory implements IFactory {
        @Override
        public AirFilter create() {
            return new AirFilter();
        }
    }
    // реализация обязательна, чтобы реализовать методы класса для Null объекта
    private static class NullAirFilter extends AirFilter implements INull {
        private NullAirFilter() { // так и так вызывается
            super();
        }
        @Override
        public String getModel() {
            return "ModelFilter.NULL";
        }
        @Override
        public String toString() {
            return "AirFilter.NULL";
        }
    }
    public Part getNull() { // INull interface
        return NULL;
    }

    @Override
    public String getModel() {
        return getClass().getSimpleName();
    }
}
```

## Динамическое применение Null

- Динамическое решение Null
  - создается в качестве статического метода в базовом классе
- **ВНИМАНИЕ.** при работе с InvocationHandler в строке Class[] { ТУТ ТОЛЬКО ИНТЕРФЕЙСЫ}
- **ВНИМАНИЕ.** объект Null создавать ТОЛЬКО ПО ТИПУ ИНТЕРФЕЙСА, ВКЛЮЧАЯ proxyObj
- хотя proxyObj можно оставить и как общий класс предок
- Пример. показана только часть Proxy
- **ВНИМАНИЕ.** Здесь показана реализация proxyObj КАК КЛАСС, НЕ как объект ИНТЕРФЕЙСА.
- ```
public class PartProxy implements InvocationHandler{
    private String nullName = "NULL";
    private Part proxyObj = new NullPart(); // создаем NullPart объект один
    private class NullPart extends Part implements Null {
        @Override
        public String getModel() {
            return "Model:NULL";
        }
        @Override
        public String toString() {
            return nullName;
        }
    }
    public PartProxy(Class <?extends Part> type) { // конструктор класса Proxy
        this.nullName = type.getSimpleName() + ".NULL"; // имя объекта
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        return method.invoke(proxyObj, args); // вызываем оригиналный метод
    }
}

public interface IPart {
    String getModel();
}

public class Part implements IPart{ // показана только часть Proxy
//getNull Object of Class
    public static IPart getNull(Class<? extends Part> type) {
        return (IPart) Proxy.newProxyInstance(
            Part.class.getClassLoader(),
            new Class[]{ IPart.class, Null.class}, // здесь только интерфейсы
            new PartProxy(type)
        );
    }
    @Override
    public String getModel() {
        return "";
    }
}
```

lesson\_ch14/ch/ex24/access

## Динамическое решение Null решение с Interface IPart

- Динамическое решение Null
  - создается в качестве статического метода в базовом классе
- **ВНИМАНИЕ.** при работе с InvocationHandler в строке Class[] { ТУТ ТОЛЬКО ИНТЕРФЕЙСЫ}
- **ВНИМАНИЕ.** объект Null создавать ТОЛЬКО ПО ТИПУ ИНТЕРФЕЙСА, ВКЛЮЧАЯ proxyObj
- **ВНИМАНИЕ.** тип IPart объекта ОБЯЗАТЕЛЬНО должен быть в качестве параметра InvocationHandler
- ```
public static IPart getNull(Class<? extends IPart> type) {
```
- 
- Пример. показана честная реализация Proxy

```
public interface IPart {
    String getModel();
}

public class Part implements IPart { // показана только часть Proxy
    private static List<IFactory<IPart>> pList = new ArrayList<>();

    public static IPart createRandom() {      // точно также создает объекты Part
        int n = rnd.nextInt(pList.size());    // случайным образом детальку из списка
        return pList.get(n).create();          // создается из типа деталька экземпляр
    }

    //getNull Object of Class
    public static IPart getNull(Class<? extends IPart> type) {
        return (IPart) Proxy.newProxyInstance(
            Part.class.getClassLoader(),
            new Class[]{ IPart.class, Null.class },           // здесь только интерфейсы
            new PartProxy(type)
        );
    }
    @Override
    public String getModel() {
        return "";
    }
}

• public class PartProxy implements InvocationHandler{
private String nullName = "NULL";
private IPart proxyObj = new NullPart(); // создаем NullPart объект один

private class NullPart implements IPart, Null {
    @Override
    public String getModel() {
        return "Model:NULL";
    }
    @Override
    public String toString() {
        return nullName;
    }
}

public PartProxy(Class <?extends IPart> type) { // конструктор класса Proxy
    this.nullName = type.getSimpleName()+"NULL"; // имя объекта
}
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    return method.invoke(proxyObj,args); // вызываем оригиналый метод
}
```

## Фиктивные объекты и заглушки

- Фиктивные объекты и заглушки
  - Mock Object    фиктивные объекты это вариант Null Object
  - легковесны и самотестируемы, в проекте несколько Mock с разным поведением
  - Stub Object    заглушка, тоже вариант Null Object
  - сложный универсальный объект, повторно используется, поведение зависит от способа вызова

## Интерфейсы и информация типов

- Интерфейсы и изоляция методов
  - интерфейсы позволяют получить унифицированный доступ к разным классам
  - интерфейсы НЕ помогают изолировать методы класса от программиста
  - приведение типа экземпляра класса интерфейса к нужному классу и методы доступны
- Пример.  
[lesson\\_ch14/ch/ex25/local](#)

```
public static void app() {
    A a = new B(); // classB interface A
    a.f();
    // a.g(); .. нужно приведение типов
    if (a instanceof B) {
        B b = (B)a;
        b.g();           // интерфейс не смог изолировать g() от программиста
    }
}
• Интерфейсы и изоляция методов с помощью package access
    ○ возможно скрыть методы класса за счет скрытия самого класса lesson\_ch14/ch/ex25/include
• ВНИМАНИЕ. reflections позволяет вытащить ВСЕ методы класса, в том числе и private access
• Пример.
class C implements A {      // уровень доступа package
    @Override
    public void f() {
    }
    public void g() {
    }
    void u() {
    }
    protected void v() {
    }
    private void w() {
    }
}
public class HiddenC {      // уровень доступа public обертка для доступа к классу C
    public static A makeA() { // получить экземпляр C статическим методом через обертку
        return new C();
    }
}
public class Include throws Exception {
    static void hiddenMethod(Object o, String name) {
        Method m = o.getClass().getDeclaredMethod(name);
        m.setAccessible(true);
        m.invoke(o); // если аргументов нет, можно без них
    }
    public static void app() {
        A a = HiddenC.makeA(); // доступ к классу <class C> через <public class HiddenC>
        a.f();
        hiddenMethod(a, "g");      // public сработало
        hiddenMethod(a, "u");      // сработало package
        hiddenMethod(a, "v");      // сработало protected
        hiddenMethod(a, "w");      // сработало private
        // if (a instanceof C) { // так как class C скрыт, не сработало
        //
    }
}
```

## Доступ к закрытым членам класса

- Доступ к закрытым методам класса
  - зная имя метода и используя метод setAccessible() можно запустить даже private методы
  - узнать имена методов класса можно при помощи декомпилиатора javap -private
- **ВНИМАНИЕ.** javap -private и метод Method.setAccessible() дают доступ к закрытым методам класса

[lesson\\_ch14/ch/ex25/include](#)

## Reflections позволяет получить доступ

- Reflections позволяет получить доступ
  - к закрытым private методам класса с доступом package
  - к закрытым private полям класса с доступом package
  - к закрытым private методам встроенного анонимного класса с доступом package
  - к закрытым полям private встроенного анонимного класса с доступом package
- Пример.
- >> javap -private out/production/lesson\_ch14/ch/ex25/include/pkg/C
- Compiled from "C.java"  

```
class ch.ex25.include.pkg.C implements ch.ex25.include.pkg.A {  
    ch.ex25.include.pkg.C();  
    public void f();  
    public void g();  
    void u();  
    protected void v();  
    private void w();  
}
```
- 

[lesson\\_ch14/ch/ex25/include](#)

[lesson\\_ch14/ch/ex25/integra](#)

## Резюме RTTI

- RTTI позволяет получить точный тип класса даже если есть ссылка с типом базового класса(предка)
- RTTI использование
- НЕ применять RTTI для доступа к методу в одном классе
  - это приведет к появлению «заглушек» в других классах
- Применять нисходящее преобразование только если
  - базовый метод неэффективно работает в каком то конкретном классе потомка, и надо получить
  - доступ к реализации метода потомка
  - это редкая ситуация, сначала надо получить рабочую программу затем думать об эффективности
- 
- 
-

## Generic Обобщенные типы

- Generic это шаблоны вместо которых подставляются реальные типы во время компиляции
  - позволяют ограничить класс хранения одним типом на уровне экземпляра при компиляции
- Пример. класс может хранить любой тип, но в данном случае только <Pet> [lesson\\_ch15/ch/ex2/access](#)
- ```
public class Holder3<T> {  
    T a;  
  
    public Holder3(T a) {  
        this.a = a;  
    }  
  
    public T getA() {  
        return a;  
    }  
  
    public void setA(T a) {  
        this.a = a;  
    }  
}  
  
public static void app() {  
    Holder3<Pet>[] h3Array = new Holder3[10];  
    for (int i = 0; i < h3Array.length; i++) {  
        h3Array[i] = new Holder3<>(Pets.randomPet());  
    }  
    for (Holder3<Pet> petHolder3 : h3Array) {  
        System.out.println(petHolder3.getA());  
    }  
}
```

## Tuple Кортежи типов

- Tuple Кортежи типов
  - конструкция <A,B,C> называется Tuple и позволяет передать в качестве Generic несколько типов
- Пример. два класса и показано как наращивать объем поддерживаемых типов
- ```
public class TwoTuple<A, B> {  
    A first;  
    B second;  
    public TwoTuple(A first, B second) {  
        this.first = first;  
        this.second = second;  
    }  
    @Override  
    public String toString() {  
        return "TwoTuple{" + first + " " + second +  
            '}';  
    }  
}  
  
public class ThreeTuple<A, B, C> extends TwoTuple {  
    C third;  
    public ThreeTuple(A a, B b, C c) {  
        super(a, b);  
        this.third = c;  
    }  
    @Override  
    public String toString() {  
        return "ThreeTuple{" + first + " " + second + " "  
            + third +  
            '}';  
    }  
}
```

## Generic встраивание в класс

- Generic встраивание в класс
  - при встраивании в класс <T> добавляют в имя класса/интерфейса
  - Т добавляют в качестве параметров входных и выходных функций
  - массивы создаются Object[] и в упрощенном варианте идет просто кастомизация Object
- Пример.
- ```
public class Sequence<T> {
    private Object[] items; // ссылка пустая на массив ссылок Object
    private int next = 0;
    private Sequence<T> pLink = this;
    public Sequence(int size) {
        items = new Object[size]; // создали массив пустых ссылок
    }
    public void add(T o) {
        if (next < items.length) {
            items[next++] = o; // прописали новую ссылку в массиве на объект
        }
    }
    public class Selector implements ISelector<T> { // inner class
        private int index = 0;
        public boolean checkEnd() { // проверяет последний это элемент или нет
            return (index >= items.length);
        }
        public T getCurrent() { // вытаскивает текущий объект
            if (checkEnd()) {
                throw new RuntimeException("Array Error");
            } else {
                return (T) items[index]; // нестрогая типизация
            }
        }
        public void nextIndex() {
            if (index < items.length) {
                index++;
            }
        }
    }
    public ISelector<T> getSelector() {
        return new Selector();
    }
}
```

## Stack Class

- Stack Class      работа со стеком и обобщенным типом
- Inner Class имеет особенности работы с Generics
  - для static Inner Class      Geneirc требуется задать тип <U>, можно отличный от внешнего <T>
  - для inner Class              внутренний класс использует тип внешнего класса <T>

## RandomList

- RandomList      список с Generic
  - в качестве типа ArrayList<T>      используется Generic
- Пример.

## Interface w Generic

- Interface w Generic используется для реализации методов с общим типом возврата
  - interface Generator<T> может возвращать как объекты <MyClass> так и <Integer>
- **ВНИМАНИЕ.** примитивы не совместимы с Generics
- Пример. генератор чисел Fibonacci
- Вариант 1.  

```
lesson_ch15/ch/ex7/include  
lesson_ch15/ch/ex7/integra
```

```
public class Fibonacci2 extends Fibonacci implements Iterable<Integer>{  
    private int count = 0;  
    private int size;  
    public Fibonacci2(int size) {  
        this.size = size;  
    }  
    private int fib(int n) {  
        if (n < 2) {  
            return 1;  
        }  
        return (fib(n-2)+fib(n-1)); // следующее число сумма двух предыдущих  
    }  
    public Integer next() {  
        return fib(count++);  
    }  
    @Override  
    public Iterator<Integer> iterator() {  
        return new Iterator<Integer>() {  
            @Override  
            public boolean hasNext() {  
                return (count < size);  
            }  
            @Override  
            public Integer next() {  
                return fib(count++);  
            }  
        };  
    }  
}  
};
```
- Вариант 2  

```
lesson_ch15/ch/ex7/integra
```

```
public class IterableFibonacci extends Fibonacci implements Iterable<Integer>{  
    private int size;  
    public IterableFibonacci(int size) {  
        this.size = size;  
    }  
    @Override  
    public Iterator<Integer> iterator() {  
        return new Iterator<Integer>() {  
            @Override  
            public boolean hasNext() { // крутим пока не переберем все элементы  
                return (size > 0);  
            }  
            @Override  
            public Integer next() {  
                size--; // получить следующий элемент  
                return IterableFibonacci.this.nextf(); // выходим на внешний класс,  
            }  
        };  
    }  
}
```

## Обобщенные методы

- Обобщенные методы это применение Generics не на уровне класса, а на уровне метода
  - class MyClass<T> Generic на уровне класса
  - <T> void myMethod(T a) Generic на уровне метода
- Пример. класс не параметризован, а метод параметризован [lesson\\_ch15/ch/ex9/local](#)
- ```
public class GMet {  
    private int count = 0;  
    public <T> void getT( T x) {  
        System.out.println(count+++"."+x.getClass().getSimpleName());  
    }  
}  
  
• public static void app() {  
    GMet gm = new GMet();  
    gm.getT(new ArrayList<>());  
    gm.getT(new HashMap<>());  
    gm.getT(new Integer(1));  
    gm.getT(new Character('1'));  
    gm.getT(1);  
    gm.getT(1L);  
    gm.getT(1F);  
    gm.getT(1.1);  
    gm.getT("Str");  
}
```

## Generic with AutoDetect

- Generic with AutoDetect автоматическое определение типа [lesson\\_ch15/ch/ex11/local](#)
  - позволяет не прописывать тип в выражении справа при присвоении
  - похоже это уже реализовано по умолчанию
- **ВНИМАНИЕ.** работает только при присвоении
- Пример. приведен пример по умолчанию, поэтому нет смысла в написании своих классов оберток
- ```
public class New {  
    public static <K,V> Map<K,V> map() {  
        return new HashMap<K, V>();  
    }  
    public static <T> List<T> list() {  
        return new ArrayList<T>();  
    }  
    public static <T> Set<T> set() {  
        return new HashSet<T>();  
    }  
    public static <T> Queue<T> queue() {  
        return new LinkedList<T>();  
    }  
  
    public static void app() {  
        // ускорение с autodetect  
        Map<Person, List<?extends Pet>> map2 = New.map();  
        List<Double> list = New.list();  
        Set<String> set = New.set();  
        Queue<Integer> queue = New.queue();  
        // по умолчанию уже работает справа ничего писать не нужно  
        Map<Person, List<?extends Pet>> hmap2 = new HashMap<>();  
        List<Double> alist = new ArrayList<>();  
        Set<String> hset = new HashSet<>();  
        Queue<Integer> queue2 = new LinkedList<>();  
        // вызов расширения как результата работы метода  
        Map<Person, List<?extends Pet>> fmap = f(New.map()); // все работает  
    }  
}
```

## Generic явное указание типа

- Generic явное указание типа
  - это указание прямого типа в методе при вызове класса оболочки

[lesson\\_ch15/ch/ex12/access](#)

## Variable length List of arguments

- Variable length List of arguments
  - списки аргументов переменной (T...) длины работают с Generic

[lesson\\_ch15/ch/ex13/local](#)

```
public class GenArgs {  
    public static <T> List<T> getList(T...args) {  
        List<T> list = new ArrayList<>();  
        for (T arg : args) {  
            list.add(arg);  
        }  
        return list;  
    }  
}  
  
public static void app() {  
    List<String> ls = GenArgs.getList("A");  
    System.out.println(ls);  
    ls = GenArgs.getList("A", "B", "C"); // предыдущий список удален GC  
    System.out.println(ls);  
}
```

## Generic и Generator

- Generic и Generator
  - создание объектов автоматом с использованием Generic
  - в интерфейсе Generator используется тип <T>
  - в классе генератора используется Collection<T>

Пример. Coffegen.next() функция генерации случайного объекта Coffee

[lesson\\_ch15/ch/ex13/include](#)

```
public class Gens {  
    public static <T> Collection <T> fill(Collection<T> col, IGenerator<T> gen, int n) {  
        for (int i = 0; i < n; i++) {  
            col.add(gen.next()); // функция реализация класса интерфейса генератора  
        }  
        return col; // выгрузить заполненную Collection  
    }  
}  
  
public static void app() {  
    Collection<Coffee> col = Gens.fill(new ArrayList<Coffee>(), new CoffeeGen(), 8);  
    System.out.println(col);  
}
```

## Basic Generator

- BasicGenerator генератор общего назначения для любого класса
  - используется в блоке try\_catch метод генерации Class.newInstance()

[lesson\\_ch15/ch/ex14/local](#)

- Пример.

```
public class GenBasic<T> implements IGenerator {  
    private Class<T> type;  
    public GenBasic(Class<T> type) {  
        this.type = type;  
    }  
    public T next() {  
        try {  
            return type.newInstance(); // создаем экземпляр заданного класса  
        } catch (Exception e) {  
        }  
    }  
    public static IGenerator<T> create(Class<T> type) { // генератор типа  
        return new GenBasic<T>(type);  
    }  
}
```

## Tuple упрощение использования

- Tuple упрощение использования
  - в общем это использование перегруженного метода с набором Generic на уровне метода
  - вручную поддерживающий все варианты количества Generic типов
- Пример.

```
public class Tuple {  
    public static <A,B> TwoTuple<A,B> tuple(A a, B b) {  
        return new TwoTuple<A,B>(a, b);  
    }  
    public static <A,B,C,D,E> FiveTuple<A,B,C,D,E> tuple(A a, B b, C c, D d, E e) {  
        return new FiveTuple<A,B,C,D,E>(a,b,c,d,e);  
    }  
}
```

[lesson\\_ch15/ch/ex15/local](#)

## Операции с Set

- Операции с Set
  - применение такое же как с Collection
- Пример.

[lesson\\_ch15/ch/ex17/local/access](#)

```
public class Sets {  
    public static <T> Set<T> union(Set<T> a, Set<T> b) { // базовая  
        Set<T> hset = new HashSet<T>(a);  
        hset.addAll(b);  
        return hset;  
    }  
    public static <T> Set<T> intersection(Set<T> a, Set<T> b) { // базовая  
        Set<T> hset = new HashSet<T>(a);  
        hset.retainAll(b); // common btw a,b  
        return hset;  
    }  
    public static <T> Set<T> difference(Set<T> a, Set<T> b) { // вспомогательная  
        Set<T> hset = new HashSet<T>(a);  
        hset.removeAll(b); // a - b вычесть общие элементы  
        return hset;  
    }  
    public static <T> Set<T> complement(Set<T> a, Set<T> b) {  
        return difference(union(a,b), intersection(a,b)); // (a+b) - (a-b)  
    }  
}
```

- Пример. клонирование EnumSet

```
public class Sets2 {  
    private static <T> Set<T> copy(Set<T> s) {  
        if (s instanceof EnumSet) {  
            return ((EnumSet)s).clone();  
        }  
        return new HashSet<>(s);  
    }  
    public static <T> Set<T> union(Set<T> a, Set<T> b) { // базовая  
        Set<T> hset = copy(a);  
        hset.addAll(b);  
        return hset;  
    }  
    public static <T> Set<T> difference(Set<T> a, Set<T> b) { // вспомогательная  
        Set<T> hset = copy(a);  
        hset.removeAll(b); // a - b вычесть общие элементы  
        return hset;  
    }  
    public static <T> Set<T> complement(Set<T> a, Set<T> b) {  
        return difference(union(a,b), intersection(a,b)); // (a+b) - (a-b)  
    }  
}
```

## Анонимные внутренние классы

- Анонимные внутренние классы
  - использование Interface <T> в анонимном классе для генерации объектов
- Пример.

```
public interface IGenerator<T> {  
    T next();  
}  
  
public class Customer {  
    private static long counter = 1;  
    private final long id = counter++;  
    public Customer() {  
    }  
    public static IGenerator<Customer> generator() { //внутренний класс в возврате  
        return new IGenerator<Customer>() {  
            public Customer next() {  
                return new Customer();  
            }  
        };  
    }  
    public String toString() {  
        return "Customer." + id;  
    }  
}
```

[lesson\\_ch15/ch/ex18/local](#)

## Применение сложных конструкций

- Применение сложных конструкций это использование наследования базовых классов
- Использование наследование ArrayList с Tuple<Generics> в качестве параметра
  - Generic используются в качестве типов хранения внутри класса Tuple внутри ArrayList
- Пример.

```
public class TupleList<A,B,C,D> extends ArrayList<FourTuple<A,B,C,D>> {  
}  
public static void app() {  
    TupleList<Coffee, Pet, String, Integer> tlist = new TupleList<>();  
    tlist.add(TupleTest.h());  
    tlist.add(TupleTest.h());  
    for (FourTuple<Coffee, Pet, String, Integer> t : tlist) {  
        System.out.println(t);  
    }  
}
```

[lesson\\_ch15/ch/ex19/local](#)

- Многоуровневая база данных на базе ArrayList

- Generic используются только в качестве генераторов

[lesson\\_ch15/ch/ex19/include/access](#)

```
public class Product {  
    private final int id;  
    private double price;  
    public Product(int id, double price) { //  
        this.id = id;  
        this.price = price;  
    }  
    public String toString() {  
        return id +", price: " + price;  
    }  
    public static IGenerator<Product> generator() {  
        return new IGenerator<Product>() {  
            private Random rnd = new Random();  
            public Product next() {  
                return new Product(rnd.nextInt(1000),  
                    Math.round(rnd.nextDouble()*1000.0)+0.99);  
            }  
        };  
    }
```

## Erasure и Generics

- Erasure и Generics [lesson\\_ch15/ch/ex20/local](#)
- Сравнение коллекций с разными типами
  - `ArrayList<String>.class == ArrayList<integer>.class >> true` несмотря на разный тип хранения
- Получение информации о типе от экземпляра класса
  - `List<Frob> list` вернет параметр [E]
  - `List<Double> list` вернет параметр [E]
- В коде с Generics работает технология Erasure которая СТИРАЕТ информацию о применяемых типах
  - в примере выше `List<String>` и `List<Integer>` стираются до низкоуровневого типа `List<>`
- **ВНИМАНИЕ.** В обобщенном коде информация о реально применяемых типах НЕ ДОСТУПНА
- Пример. [lesson\\_ch15/ch/ex20/local](#)

## Bounds Ограничения

- Ограничения это указание компилятору работать в рамках группы классов
  - стирают тип до класса ограничения `<T extends ClassLimiter>`
  - Generic уже не работают
  - только для Generic можно заменить `<T extends ClassName> >> <ClassName>` МОЖНО
  - если есть методы `<T>` заменять `<T extends ClassName> >> <ClassName>` НЕЛЬЗЯ
- Пример. код без ограничений типа `<T>` [lesson\\_ch15/ch/ex20/local](#)

```
public class HasF {           // класс HasF имеет метод f()
    public void f() {
    }
}
public class Manipulator<T> {
    T obj;
    public Manipulator(T obj) {
        this.obj = obj;
    }
    public void manipulator() {
//        obj.f() // компилятор метод не находит
    }
}
```
- Исправленная версия с внесением ограничения `<T extends HasF>`
  - то есть тип T имеет доступ к методам HasF как минимум
- Пример.
- ```
public class Manipulator2<T extends HasF> {
    T obj;
    public Manipulator2(T obj) {
        this.obj = obj;
    }
    public void manipulator() {
        obj.f();           //метод обнаружен
    }
}
```
- **ВНИМАНИЕ.** Ограничения СТИРАЮТ тип до класса ОГРАНИЧЕНИЯ это значит GENERIC уже НЕ РАБОТАЮТ
- **ВНИМАНИЕ.** конструкцию `<T extends Class>` МОЖНО заменить на `<Class>` если дело только в Generics
- **ВНИМАНИЕ.** конструкцию `<T extends Class>` НЕЛЬЗЯ заменить на `<Class>` если есть методы с типом `<T>`
- 
- 
- 
- 
-

## Миграционная совместимость

- Миграционная совместимость
  - стирание создано для совместимости обобщенного класса с точными типами в библиотеках
  - то есть обобщенные типы позволяют совместимость при миграции библиотек
- **ВНИМАНИЕ.** СТИРАНИЕ это переход от точного кода к обобщенному и возможность встраивания обобщений в язык без конфликта с существующими библиотеками
- СТИРАНИЕ позволяет работать клиентскому коду с точными типами до тех пор, пока не потребуется переписать на обобщения

[lesson\\_ch15/ch/ex21/local](#)

## Потеря объекта при наследовании

- Потеря объекта при наследовании
  - при наследовании и отсутствии конструктора с параметром можно потерять объект
  - и пропустить инициализацию , что приведет к исключению только в RunTime()
- **ВНИМАНИЕ.** Если есть поля объекты в классе предке СОЗДАВАТЬ конструктор с параметрами у ПРЕДКА
- Пример. поле element теряется при создании экземпляров потомков т.к. нет Конструктора у ПРЕДКА
- ```
public class GenericBase<T> { //класс предок конструктора для поля element нет обнаружен
    private T element;
    public T getElement() {
        return element;
    }
    public void setElement(T element) {
        this.element = element;
    }
    public void getInfo() {
        System.out.println("element:" + element.getClass().getSimpleName());
    }
}
public class Derived1 <T> extends GenericBase<T> { // класс потомок первый уровень
}
public class Derived2<T> extends Derived1<T> { // класс потомок второй уровень
}
public static void app() {
    Derived2<Cat> d2 = new Derived2<>(); // компилируется никаких предупреждений
    d2.getInfo(); // NullPointerException во время RunTime
}
```

[lesson\\_ch15/ch/ex21/include](#)

## Границные ситуации Action at the boundaries

- Границные ситуации
  - использование обобщений может привести к бессмысленным ситуациям
  - можно получить массив null, хотя формально это массив строк
- Пример.
- ```
public class ArrayMaker<T> {
    private Class<T> kind;
    public ArrayMaker(Class<T> kind) {
        this.kind = kind;
    }
    T[] create(int size) {
        return (T[]) Array.newInstance(kind, size); // создать массив объектов
    }
}
```

[lesson\\_ch15/ch/ex21/integra](#)

## Недостатки Generic

- Стирание порождает ограничения Generic это плата за универсальность
  - Generic не работают с приведением типа
  - Generic не поддерживают конструкции <instanceof>
  - Generic не очень хорошо работают с new()
- **ВНИМАНИЕ.** Любые операции во время RunTime которые требуют знания точного типа НЕ РАБОТАЮТ
- Пример.  
[lesson\\_ch15/ch/ex21/remote](#)
- ```
public class Erased<T> {
    private static final int SIZE = 100;
    public static void f(Object arg) {
        if (arg instanceof T) { // не работает
        }
        T[] var      = new T(); // не работает
        T[] vArray   = new T[SIZE]; // не работает
        T[] vArray2 = (T[]) new Object[SIZE]; // не работает
    }
}
```

## Компенсация стирания

- Компенсация стирания  
[lesson\\_ch15/ch/ex21/remote](#)
  - чтобы обойти ограничения стирания используют метку класса и функцию isInstance()
- Пример. определение типа Generic класса через метку класса
- ```
public class ClassCapture<T> { // класс захвата типа класса
    Class<T> kind;
    public ClassCapture(Class<T> kind) {
        this.kind = kind;
    }
    public boolean f(Object arg) {
        return kind.isInstance(arg); // <arg is instance of kind>
    }
}
public static void app() {
    ClassCapture<House> ctt2 = new ClassCapture<>(House.class); // образец класса
    System.out.println("Building instanceof House : "+ctt2.f(new Building()));
    System.out.println("House instanceof House : "+ctt2.f(new House()));
}
```

## Создание экземпляров типов

- Создание экземпляров типов
- Создание new T() не работает
  - из за стирания компилятор не знает какой тип
  - из за отсутствия информации о конструкторе по умолчанию для <T>
- Решение проблемы создания экземпляра через метку класса  
[lesson\\_ch15/ch/ex22/local](#)
  - используется Class<?> kind метка класса
  - метод создания экземпляра Class.newInstance()
- **ВНИМАНИЕ.** Такой способ выдает ошибку только при RunTime что НЕ РЕКОМЕНДУЕТСЯ
- Решение проблемы с явным указанием типа  
[lesson\\_ch15/ch/ex22/include](#)
  - общий смысл, создавать объекты при помощи явно созданной фабрики на базе IFactory
  - так отсекаются встроенные классы которые не поддерживают IFactory на этапе компиляции
- Решение на базе паттерна «Шаблонный метод»
  - наследует класс обертку, который использует IFactory и подставляет точный тип в потомке
  - по сути проще использовать IFactory напрямую с точным типом

- Создание экземпляра с помощью конструктора
  - **ВНИМАНИЕ.** есть ограничение на использование примитивных типов и reflections

- Пример.

```

public class Ball {
    private String color;
    private int size;

    public Ball(String color, int size) {
        this.color = color;
        this.size = size;
    }
    public String toString() {
        return "Ball{" + color + ", " + size + '}';
    }
}

public class ClassFactory<T> {
    T x;   // экземпляр Generic класса
    Class<T> kind;
    public ClassFactory(Class<T> kind) {
        this.kind = kind; // Class label
    }
    private boolean check(Parameter p, String name) {
        if (p == null) {
            return false;
        }
        return name.equals(p.getParameterizedType().get TypeName() .
            replaceAll("\\w+\\.\\.", ""));
    }
    public <T,U,V> T create(U s, V i) {
        try {
            for (Constructor<?> con : kind.getConstructors()) {
                Parameter [] p = con.getParameters();
                if (p.length == 2 && check(p[0], s.getClass().get SimpleName())) {
                    return (T) con.newInstance(s, i);
                }
            }
        } catch (Exception e) {
            throw new RuntimeException("catch: Can't create Instance"); //Exception
        }
        throw new RuntimeException("Constructor not Found"); // вызываем Exception
    }
}

```

- Создание генератора с аргументом

[lesson\\_ch15/ch/ex23/access](#)

- Пример.

```

public interface IFactory2<T,V> {
    T create(V v);
}

public class Foo2 <T,V> {
    private T x;
    public <F extends IFactory2<T,V>> Foo2(F object,V v) {
        x = object.create(v);
    }
    public T getX() {
        return x;
    }
}

public class IntegerFactory implements IFactory2<Integer, Integer> {
    public Integer create(Integer integer) {
        return new Integer(integer);
    }
}

```

## Массивы обобщений

- Массивы обобщений
  - массив с Generic можно создать на базе ArrayList<T>
  - массив с Generic можно создать на базе массива класса обертки ClassName<T>
  - массив с Generic можно создать на базе массива Object [] с кастомизацией (T[])new Object[size]
  - массив с Generic создать нельзя, так как Generic не поддерживают оператор new T[]
- **ВНИМАНИЕ.** Во время компиляции массив содержит тип ClassName<T> но в RunTime это массив Object
- Generic Array[] работает как Object во время Runtime
  - это означает, что как уже говорилось, все типы стираются
  - и в массив созданный как <Integer> во время RunTime можно писать как в <String>
- **ВНИМАНИЕ.** Рекомендуется создавать Generic[] как Object[], приведение <T> делать на уровне элементов
- Пример.

[lesson\\_ch15/ch/ex25/local](#)

```
public class GenericArray<T> {  
    private T[] array = (T[])new Object[10];  
    public GenericArray(int size) {  
        this.array = (T[])new Object[size];  
    }  
    public void put(int index, T item) {  
        array[index] = item;  
    }  
    public T get(int index) {  
        return array[index];  
    }  
    public T[] rep() { // выдаем ссылку на массив внутри класса  
        return array;  
    }  
}  
  
public static void app() {  
    GenericArray<Integer> genIntArray = new GenericArray<>(10);  
    try {  
        Integer[] intArray = genIntArray.rep(); // cast не прокатывает  
    } catch (Exception e) {}  
    Object[] objArray = genIntArray.rep(); // вот так работает  
    for (int i = 0; i < objArray.length; i++) {  
        objArray[i] = "str"+i; // прокатило, то есть от Integer ничего не осталось  
    }  
    System.out.println("Object Array");  
    for (Object o : objArray) {  
        System.out.print(o+" ");  
    }  
}
```

[lesson\\_ch15/ch/ex25/include](#)

- Приведение типа на уровне элементов

- создать массив в классе<T> типа Object[]
- читать элементы с приведением типа (T)array[index]

```
public class GenericArray2<T> {  
    private Object[] array = new Object[10]; // рекомендованная стратегия  
    public void put(int index, T item) {  
        array[index] = item;  
    }  
    public T get(int index) {  
        return (T) array[index];  
    }  
    public Object[] rep() { // выдаем ссылку на массив внутри класса  
        return array;  
    }  
}
```

- Применение метки класса для усиленного контроля массива Generic[] [lesson\\_ch15/ch/ex25/integra](#)
  - использование функции Array.newInstance(T[]) позволяет сохранить тип массива в RunTime
- **ВНИМАНИЕ.** Приведение типа массива (T[]) Object[] является НЕБЕЗОПАСНЫМ
- **ВНИМАНИЕ.** ArrayList класс использует НЕБЕЗОПАСНЫЕ преобразования, соблюдать осторожность с ним
- Пример. создание массива который поддерживает типизацию в RunTime
- ```
public class GenArrayToken<T> {
    private T[] array;
    public GenArrayToken(Class<T> type, int size) {
        array = (T[])Array.newInstance(type, size);
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) {
        return array[index];
    }
    public T[] rep() {
        return array;
    }
}
public static void app() {
    GenArrayToken<Integer> gt = new GenArrayToken<>(Integer.class, 10);
    Integer[] pt = gt.rep();
    System.out.println("GenArrayToken rep(): done");
}
```

## Bounds Ограничения

- Bounds Ограничения
  - ограничения это сужение поддерживаемых типов до одного класса
  - позволяют компилятору сузить Generic до класса, получить информацию о методах и полях класса
  - без ограничений компилятору доступны только информация о методах и полях класса Object
- Синтаксис
  - <T extends MyClass> ограничение типа используемого класса до класса MyClass
  - компилятору гарантированно доступны методы, поля класса MyClass
- **ВНИМАНИЕ.** <T extends ClassA & IFaceA & IFaceB> только так, класс первым, затем интерфейсы через &
- Пример. [lesson\\_ch15/ch/ex25/remote](#)

## Bounded компактная запись

- Bounded компактная запись
  - при наследовании каждый уровень добавляет ограничения
- Пример. Bounded компактная запись базовые интерфейсы и классы
 

```
public interface IColor {
    Color getColor(); // метод возвращает тип Colored
}
public interface IWeight {
    int getWeight();
}
public class Dimension {
    public int x,y;
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

- Пример. Bounded компактная запись продолжение, классы хранения [lesson\\_ch15/ch/ex25/remote](#)

```

• public class Bounded extends Dimension implements IColor, IWeight{
    @Override
    public int getWeight() {
        return 0;
    }
    @Override
    public Color getColor() {
        return null;
    }
}

• public class HoldItem<T> {
    T item; // T поддерживает только Object
    public HoldItem(T item) {
        this.item = item;
    }
    public T getItem() {
        return item;
    }
}

• public class Colored<T extends IColor> extends HoldItem<T> { // конструктор от
super
// T item // поддерживает теперь интерфейс IColor
    public Colored(T item) {
        super(item);
    }
    Color getColor() {
        return item.getColor();
    }
}

• public class ColoredDim<T extends Dimension & IColor> extends Colored<T> {
//T item; // добавлено новое ограничение поддержка Dimension
    public ColoredDim(T item) {
        super(item);
    }
    int getX() { // Dimension
        return item.getX();
    }
    int getY() {
        return item.getY();
    }
}

• public class Solid <T extends Dimension & IColor & IWeight> extends ColoredDim<T>{
// T item; // добавлено ограничение поддержка IWeight
    public Solid(T item) {
        super(item);
    }
    int getWeight() { // IWeight
        return item.getWeight();
    }
}

```

## Wildcards Маски

- Маски это использование wildcard в объявлении типов классов
  - <T extends MyClass> это объявление любого типа <T> который является потомком класса MyClass

[lesson\\_ch15/ch/ex26/local](#)

## Массивы

- Массивы при создании четко фиксируют тип элементов и происходит это в два этапа
  - первый этап присвоение здесь честно работает восходящее преобразование Fruit[] = Apple[]
  - второй этап использование теперь это уже массив Apple[] и только Apple, Fruit[] уже нет
- **ВНИМАНИЕ.** Восходящее преобразование при присвоении работает как обычно, но в результате останется массив потомков как будто он изначально создавался как массив потомков
- Fruit[] = new Apple[] массив позволяет создать в теле Fruit[] объекты Apple()
- **ВНИМАНИЕ.** Восходящее преобразование при использовании работает ТОЛЬКО для реального типа элементов массива, то есть тем элементам которые были в операторе new Apple[]
- Но принимать этот массив будет ТОЛЬКО объекты Apple или потомков Apple НЕ Fruit

- Пример.

```
• public class Fruit {  
}• public class Apple extends Fruit {  
}• public class Jonathan extends Apple {  
}• public class Orange extends Fruit {  
}• public static void app() {  
    Fruit[] fruits = new Apple[10]; // восходящее Fruit только при присвоении  
                                // пока это пустой массив и это уже Apple[]  
    for (int i = 0; i < fruits.length; i++) {  
        fruits[i] = new Apple(); // просто заполняем  
    }  
    fruits[5] = new Jonathan(); // восходящее Apple работает  
    fruits[7] = new Jonathan(); // восходящее Apple работает  
    try {  
        fruits[3] = new Fruit(); // массив заявлен как Apple not Fruit Runtime Ex  
    } catch (Exception e) {  
    }  
    try {  
        fruits[3] = new Orange(); // массив заявлен как Apple not Orange Runtime  
Ex  
    } catch (Exception e) {  
    }  
    for (Fruit fruit : fruits) {  
        System.out.println(fruit);  
    }  
}
```

## List

- List при создании с восходящим преобразованием вообще не работает
  - выдает несовместимые типы при назначении List<Fruit> = new ArrayList<Apple>  
// List<Fruit> fruits = new ArrayList<Apple>(); // не компилируется
  - компилируется, но не заполняется при назначении List<? extends Fruit> = new ArrayList<Apple>
- **ВНИМАНИЕ.** Компилятор не добавляет, потому что не знает какой точно тип разрешено добавить
- **ВНИМАНИЕ.** Wildcard позволяет компилятору обойти ограничение типа, но оно же не дает работать List<? extends Fruit> fList = new ArrayList<Apple>(); // компилируется

- //            flist.add(new Apple());    // не компилируется  
   //            flist.add(new Orange());    // не компилируется  
   //            flist.add(new Fruit());    // не компилируется

## Class User and Wildcards

- Class User and wildcards       работает точно также как List<?extends MyClass>
  - wildcard позволяет создать класс хранения, но работает для изменения элемента хранения
  - wildcard работает на уровне ссылки на объект класса хранения
- **ВНИМАНИЕ.** конструкция <?extends MyClass> НЕ РАБОТАЕТ в полной мере
  - wildcard работает НА СРАВНЕНИЕ и ИЗВЛЕЧЕНИЕ
  - wildcard НЕ РАБОТАЕТ на ЗАПИСЬ И ИЗМЕНЕНИЕ
- Пример. [lesson\\_ch15/ch/ex28/local](#)
- ```
public class Holder<T> {
    private T value;
    public Holder(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
    public void setValue(T value) {
        this.value = value;
    }
    @Override
    public boolean equals(Object o) {
        return value.equals(o);
    }
}
public static void app() {
    Holder<Apple> hApple = new Holder<>(new Apple());           // тип хранения Apple
    Holder<Orange> hOrange = new Holder<>(new Orange());         // тип хранения
    Orange
    Apple apple = hApple.getValue();          // получаем элемент из массива
    Orange orange = hOrange.getValue();
    hApple.setValue(apple);                  // работает возвращаем
    // Holder<Fruit> hFruit = hApple;          // не работает так как разный тип

    Holder<?extends Fruit> hFruit = hApple;           // работает так как wildcard
    Fruit fruit = hFruit.getValue();                 // вытащить фрукт по ссылке
    apple = (Apple)hFruit.getValue();
    // Orange orange = (Orange)hFruit.getValue();       // ломаем типы не работает
    // hFruit.setValue(new Apple()); // не работает
    // hFruit.setValue(new Fruit()); // не работает
    System.out.println("hFruit<>apple:" + hFruit.equals(apple)); // работает
    hFruit = hOrange;
    System.out.println("hFruit<>orange:" + hFruit.equals(orange)); // работает
}
```

## Wildcards решение

- Ограничение супертипа       это ограничение по базовому классу для данного массива
  - <? super MyClass> означает в коллекцию можно добавить класс MyClass и его потомков
- Синтаксис [lesson\\_ch15/ch/ex28/include](#)
  - <? super MyClass>
  - <? super T>
- **ВНИМАНИЕ.** <T super MyClass> не работает



- Пример.

[lesson\\_ch15/ch/ex28/integra](#)

```

public static <T extends Fruit> void writeTo(List<? super Fruit> list, T obj) {
    list.add(obj);
}
public static <T> void writeExact(List<T> list, T obj) {
    list.add(obj);
}
public static <T> void writeWildCard(List<? super T> list, T obj) {
    list.add(obj);
}
public static void app() {
    List<? super Fruit> list = new ArrayList<>();
    writeTo(list, new Orange()); // basic Class Fruit
    writeExact(list, new Apple());
    writeWildCard(list, new Fruit());

    List<Apple> listApple = new ArrayList<>();
    writeExact(listApple, new Apple());
    writeWildCard(listApple, new Apple());

    List<Fruit> listFruit = new ArrayList<>();
    writeTo(listFruit, new Apple()); // basic Class Fruit
    writeExact(listFruit, new Orange());
    writeWildCard(listFruit, new Fruit());
}

```

## Анализ ограничений

- Анализ ограничений Bounds и WildCard <? super T>
- Пример.

[lesson\\_ch15/ch/ex28/remote](#)

```

public class Remote {
    static List<Apple> apples = new ArrayList<>();
    static List<Fruit> fruits = new ArrayList<>();
    public static <T> void writeExact(List<T> list, T item) {
        list.add(item);
    }
    public static <T> void writeWildCard(List<?super T> list, T item) {
        list.add(item);
    }
    public static void f1() {
        writeExact(apples, new Apple());
//        writeExact(apples, new Orange()); // не работает
        writeExact(fruits, new Apple()); // работает
        writeExact(fruits, new Orange()); // работает
        writeExact(fruits, new Fruit()); // работает
    }
    public static void f2() {
        writeWildCard(apples, new Apple());
//        writeWildCard(apples, new Orange()); // не работает
        writeWildCard(fruits, new Apple()); // работает
        writeWildCard(fruits, new Orange()); // работает
        writeWildCard(fruits, new Fruit()); // работает
    }
    public static void app() {
        f1();
        f2();
        System.out.println("apples:"+apples);
        System.out.println("fruits:"+fruits);
    }
}

```

-

## **Анализ ограничений**

- Анализ ограничений Bounds и Wildcard <? extends T>
- Анализ ограничений Bounds и Wilcard <? super T >

[lesson\\_ch15/ch/ex28/remote](#)  
[lesson\\_ch15/ch/ex28/remote](#)

## **Неограниченные маски**

- Неограниченные маски <?> позволяют использовать любой класс без ограничений
  - это работает для List
  - это работает для Map
- **ВНИМАНИЕ.** В современной версии Java нет разницы «не применять», применять <?>, <?extends Object>

## **Generics vs Unbounded Mask**

- Generics vs Unbounded Mask когда и как различаются
- **ВНИМАНИЕ ПРОВЕРИТЬ РАЗЛИЧИЕ ПОХОЖЕ АБЗАЦ УСТАРЕЛ ????**

[lesson\\_ch15/ch/ex29/include](#)

## **Capture ConversionФиксация**

- Capture Conversion Фиксация
  - если передать Generic методу с указанием <?> компилятор может определить точный тип
  - такой механизм называется Capture Conversion
- **ВНИМАНИЕ.** Этот механизм АВТОМАТОМ включен

[lesson\\_ch15/ch/ex29/integra](#)

## Проблемы и решения Java Generic

### Generic не понимают примитивные типы

- Проблема невозможна создать массив `ArrayList<int>` [lesson\\_ch15/ch/ex30/access](#)
- Решение применить класс обертку `ArrayList<Integer>`
  - классы обертки есть для всех примитивов
  - заполнение происходит через автоматическую упаковку
- **ВНИМАНИЕ.** Generic `<T>` для массивов `T[]` не поддерживают `int[], char[]` или другие primitive

Пример.

```
<T> T[] fill(T[] array, IGenerator<T> gen) {
    for (int i = 0; i < array.length; i++) {
        array[i] = gen.next();
    }
    return array;
}
public void appArrayInt() {
    System.out.println("Array of <int>:");
    // int[] intA = new int[10];
    // int[] ints = fill(intA,new RandInt());           // primitive не работает с <T>
    Integer[] ints = fill(new Integer[10],new RandGen()); // Integer работает с <T>
    for (Integer anInt : ints) {
        System.out.print(anInt+" ");
    }
}
```

### Generic Interfaces

- Проблема класс не понимает две реализации интерфейса `<T>` с разным типом в ветке наследования
- Решение использовать разные имена интерфейса [lesson\\_ch15/ch/ex31/local](#)
- Пример

```
public class Employee implements Payable<Employee> {
    @Override
    public void payCheck(Employee arg) {
    }
}
public class Technics extends Employee implements Payable<Technics>{
    @Override
    public void payCheck(Technics arg) { // не компилируется так как разный тип
        // аргумента одного и того же метода
    }
}
```

- **ВНИМАНИЕ.** Если идет наследование классов и наследование интерфейсов то ДОПУСТИМО переопределение метода в цепи наследования правда типы определены ТОЧНО

- Пример. [lesson\\_ch15/ch/ex31/include](#)

```
public interface Payable {
    void payCheck(Employee arg);
}
public interface Payable2 extends Payable {
    void payCheck(Technics arg); // переопределенный метод
}
public class Employee implements Payable { // наследование классов
    @Override
    public void payCheck(Employee arg) {
    }
}
public class Technics extends Employee implements Payable2 { //наследование
    //интерфейса
    @Override
    public void payCheck(Technics arg) {
```

```
}
```

## Cast and InstanceOf

- Проблема Cast and instanceOf не работают с Generics
- Решение использовать Class.cast() нового типа
- **ВНИМАНИЕ.** Похоже все решено автоматом
- Пример.

```
public static <T> List<T> f() {
    String[] args = new String[]{"", "", "", "", "", ""}; // пустые аргументы
    List<T> list = new ArrayList<>();
    List <T> listNew = new ArrayList<T>();
    try {
        ObjectInputStream in= new ObjectInputStream(new FileInputStream(args[0]));
        list = (List<T>)in.readObject(); // работает, по старому типу
        listNew = List.class.cast(in.readObject()); // работает по новому типу
    } catch (Exception e) {
    }
    return listNew; // возврат по новому типу
}
```

[lesson\\_ch15/ch/ex32/local](#)

## Перегрузка

- Проблема Перегрузка не работает с Generic
- Решение Изменить имена функций

[lesson\\_ch15/ch/ex34/local](#)

## Перехват интерфейса базовым классом

- Перехват интерфейса базовым классом
- Проблема Если базовый класс задал тип <T> то потомки не могут это изменить
- Решение Использовать у наследников ДРУГОЙ интерфейс с ДРУГИМИ именами методов
- Пример

```
public class Pet2 implements Comparable<Pet> {
}
// public class Cat2 extends Pet2 implements Comparable<Cat> { // так не работает
// }
public class Cat2 extends Pet2 implements Comparable<Pet> { // работает
}
```

[lesson\\_ch15/ch/ex34/include](#)

## SelfBound Самоограничивающиеся типы

- Проблема SelfBound типы
  - Теория
  - Конструкция `class SubHolder extends BasicHolder<SubHolder>`
    - означает, что данный класс расширяет базовый, но тип параметра в базовом будет типом потомка
    - и методы базового класса будут работать только с типом параметра потомков
    - по сути означает что базовый класс содержит методы которые работают только с потомками
  - Конструкция `public class SelfBound<T extends SelfBound<T>>`
    - означает что в классе есть параметр, тип которого может быть только типом самого класса
    - и другие типы туда просто не проходят
    - `public class ClassA extends SelfBound<ClassA>`
    - означает что в экземпляре ClassA будет параметр типа ClassA и никакой другой
  - Для чего это нужно?
  - Если скажем таких классов много ClassA, ClassB, ClassC и они наследуют SelfBound класс и его методы
    - методы SelfBound будут как шаблон применены ко всем классам потомкам
    - параметры для методов шаблонов будут строго ограничены классами потомками
    - невозможно воспользоваться методом предка если параметр не является потомком (ClassD)
  - **ВНИМАНИЕ.** Самоограничение нужно для ограничения параметров метода предка классами потомками
  - Пример.
  - ```
public class SelfBound<T extends SelfBound<T>> {
    T element;
    public void set(T arg) {
        element = arg;
    }
    public T get() {
        return element;
    }
}
public class ClassA extends SelfBound<ClassA> { //работает ПОТОМOK и сам же аргумент
}
public class ClassB extends SelfBound<ClassA> { // сам ПОТОМOK доступ к методам
} // аргумент чужой но ПОТОМOK
public class ClassD { // класс не ПОТОМOK SelfBound
}
//public class ClassE extends SelfBound<ClassD>{// не работает сам ПОТОМOK,
аргумент
//} // ClassD не ПОТОМOK
```

## Ковариантность аргументов

- Проблема Ковариантность аргументов
- Наследуемый метод может перегрузить метод предка
- Пример. перегрузка метода

[lesson\\_ch15/ch/ex34/remote](#)

```
public class OrdinarySetter {  
    void set(Base obj) {  
    }  
}  
public class DerivedSetter extends OrdinarySetter{  
    void set(Derived obj) { // перегрузка метода  
    }  
}  
public static void app() {  
    DerivedSetter ds = new DerivedSetter();  
    ds.set(new Base()); // вызывается метод set() предка  
    ds.set(new Derived()); // вызывается метод set() потомка  
}
```

- Наследуемый SelfBound метод используется только один метод с аргументом потомком
  - за счет самоограничения остается только один метод который работает с потомками
- Пример. SelfBound Interface

[lesson\\_ch15/ch/ex34/remote](#)

```
public interface ISelfBoundSetter<T extends ISelfBoundSetter<T>> {  
    void set(T arg); // T тип самоограничивающий интерфейс  
}  
public interface ISetter extends ISelfBoundSetter<ISetter> {  
    // тоже самое, ISetter ПОТОМOK метод set(T) где T ISetter в предке  
    ISelfBoundSetter  
}  
public static void app() {  
    ISetter s1 = new Setter();  
    ISetter s2 = new Setter();  
    ISelfBoundSetter sb = new Setter(); // по предку создаем экземпляр  
    s1.set(s2); // прокатывает по интерфейсу ISetter  
    // s2.set(sb); // не работает  
}
```

- Пример. Not SelfBount Interface
- **ВНИМАНИЕ.** НЕ СОВСЕМ КОРРЕКТНЫЙ ПРИМЕР сначала говорилось об интерфейсах

затем подсовывается базовый класс <T> с типом <Base>

```
public class GenericSetter<T>{  
    public void set(T arg) {  
    }  
}  
public class DerivedGS extends GenericSetter<Base> {  
    public void set(Derived arg) {  
    }  
}  
public static void app() {  
    DerivedGS dgs = new DerivedGS();  
    dgs.set(new Derived()); // прямое применение точного типа  
    dgs.set(new Base()); // базовый класс с <T>= <Base> кривая перегрузка  
}
```

## Динамическая безопасность типов

- Проблема при передаче контейнеров с обобщенным типом контейнерам с точным типом
- Решение проверка типов методами Collections
  - checkedCollection(), checkedList(), checkedSet(), checkedSortedMap, checkedSortedSet()
- Пример. применение функции CheckedList для контроля типов в RunTime [lesson\\_ch15/ch/ex35/local](#)

```
static void oldStyle(List list) { // метод старого стиля без
    list.add(new Cat()); // указания объекта хранения
}
public static void app() {
    List<Dog> dogs = new ArrayList<>();
    dogs.add(new Pug());
    dogs.add(new Mutt());
    oldStyle(dogs); // добавляет в список <Dog> объект <Cat>

    List<Dog> dogs2 = Collections.checkedList(new ArrayList<Dog>(), Dog.class);
    dogs2.add(new Pug());
    dogs2.add(new Mutt());

    try {
        oldStyle(dogs2); // срабатывает exception не пропускает <Cat>
    } catch (Exception e) {
    }

    List<Pet> pets = Collections.checkedList(new ArrayList<Pet>(), Pet.class);
    pets.add(new Pug());
    pets.add(new Cymr()); // работает по восходящему преобразованию
}
```

## Exceptions Исключения

- Проблема Из за стирания работа с Exceptions ограничена
    - catch может перехватить только обобщенные Exception
    - Generic не может наследовать Throwable
  - Решение Использовать параметры типы в секции <throws> [lesson\\_ch15/ch/ex36/local](#)
  - Мощный пример использования Exception вместе с Generic <T>
  - Описание. Интерфейс использует два типа, класс хранения <T> и класс Exception <E>
    - <T,E extends Exception> означает <T,E> где E это расширение Exception
    - IProcessor<T,E> интерфейс на один метод processor(List<T>)
  - Processor1 {} класс реализации интерфейса <T>=<String>, <E> = Exception1
  - Processor2{} класс реализации интерфейса <T>=<Integer>, <E>= Exception2
  - ProcessorRunner{}
    - класс расширяет ArrayList с типом хранения IProcessor<T,E>
    - при создании заполняется экземплярами IProcessor<T,E> new Processor1(), new Processor2()
    - обработчик processorAll() прогоняет экземпляры списка и отрабатывает реализацию process()
  - Пример. Интерфейс и классы хранения
- ```
public class Failure1 extends Exception{
}
public class Failure2 extends Exception{
}
```

- ```
public interface IProcessor<T,E extends Exception> {
    void process(List<T> list) throws E;
}
```

- Пример. Классы реализации Processor1, Processor2, ProcessorAll

```

public class Processor1 implements IProcessor<String, Failure1> {
    static int count = 3; // T=String E=Failure1
    @Override
    public void process(List<String> list) throws Failure1 {
        if (count-- > 1) {
            list.add("Help!");
        } else {
            list.add("Ho!");
        }
        if (count < 0) {
            throw new Failure1(); // три раза отработали
        }
    }
}

public class Processor2 implements IProcessor<Integer, Failure2> {
    static int count = 2; // T=Integer E=Failure2
    @Override
    public void process(List<Integer> list) throws Failure2 {
        if (count-- > 1) {
            list.add(15);
        } else {
            list.add(10);
        }
        if (count < 0) {
            throw new Failure2(); // три раза отработали
        }
    }
}

public class ProcessorRunner<T,E extends Exception> extends
ArrayList<IProcessor<T,E>> {
    List<T> processAll() throws E {
        List<T> list = new ArrayList<>(); // Тип хранения список обработку
        for (IProcessor<T,E> iProc : this) { // расширяет ArrayList типа IP<T,E>
            iProc.process(list); // аргумент processor(List<T>)
        } // прогнать все объекты this
        return list; // вернули список
    }
}

```

- Применение.

```

public static void app() {
    ProcessorRunner<String, Failure1> ps = new ProcessorRunner<>();
    for (int i = 0; i < 3; i++) {
        ps.add(new Processor1()); // три объекта Processor1
    }
    try {
        System.out.println(ps.processAll());
    } catch (Exception e) {
        System.out.println(e);
    }
    ProcessorRunner<Integer, Failure2> ps2 = new ProcessorRunner<>();
    for (int i = 0; i < 2; i++) {
        ps2.add(new Processor2()); // два объекта Processor2
    }
    try {
        System.out.println(ps2.processAll());
    } catch (Exception e) {
        System.out.println(e);
    }
}

```

## Mixing Примеси

- Проблема Mixing это примеси методов базовых классов, в Java нельзя наследовать <T> класс
- Решение Mixing с через интерфейсы и делегирование методов классов, реализующих интерфейсы
- Решение. Mixing Decorator, ограниченное решение
- Решение Mixing Proxy подмешивает методы, но требует нисходящее преобразование
- Теория
  - есть интерфейсы IA и IB у каждого набора методов
  - есть классы A и B которые реализуют интерфейсы A =>IA, B=>B
  - чтобы пристегнуть реализации к данному классу C используется композиция и реализация IA, IB
  - класс C содержит объекты A и B встроенные через интерфейсы IA и IB
  - так как класс C должен реализовать интерфейсы IA, IB создаются методы оболочки
  - которые делегируют исполнения методам экземпляров A и B и т.к. те встроены через IA, IB
  - гарантирована совместность при делегировании и отсутствие лишних методов из классов A,B

[lesson\\_ch15/ch/ex37/access](#)

- Пример.

[lesson\\_ch15/ch/ex37/local](#)

```
• public interface ITimeStamp {
    long getStamp();
}

• public interface ISerialNumber {
    long getSerialNumber();
}

• class TimeStamp implements ITimeStamp {
    private final long timeStamp;
    public TimeStamp() {
        timeStamp = Calendar.getInstance().getTime().getTime();
    }
    public long getStamp() {
        return timeStamp;
    }
}

public class SerialNumber implements ISerialNumber{
    private static long count = 1;
    private final long sNumber = count++;
    @Override
    public long getSerialNumber() {
        return sNumber;
    }
}

public class BasicMixing extends Basic implements ITimeStamp,ISerialNumber {
    private ITimeStamp t = new TimeStamp();
    private ISerialNumber s = new SerialNumber();
    @Override
    public long getStamp() {
        return t.getStamp(); // вытаскиваем готовую реализацию через делегирование
    }
    public long getSerialNumber() {
        return s.getSerialNumber(); // делегирование реализации метода
    }
}
```

- Применение. Класс BasicMixing подмешивает к себе методы <ITimeStamp>,<ISerialNumber> с реализацией

```
public static void app() {
    BasicMixing bm1 = new BasicMixing();
    BasicMixing bm2 = new BasicMixing();
    System.out.println("bm1:"+" time:"+bm1.getStamp()+""
sn:+bm1.getSerialNumber());
    System.out.println("bm2:"+" time:"+bm2.getStamp()+""
sn:+bm2.getSerialNumber());
```

```
    sn: "+bm2.getSerialNumber());
}
```

## Mixing и Паттерн Decorator

- Декоратор делает примесь за счет переопределения метода базового класса потомками
- Проблема      Тип объекта всегда будет типом класса верхнего уровня,  
                      в одном уровне видны методы только одного класса верхнего уровня
- Решение.      Mixing Decorator ограниченное решение
- **ВНИМАНИЕ.** По моему совершенно дебильное решение
  - неявный вызов методов декоратора
  - скрытый механизм расширения функционала
  - вызывается только при инициализации объекта, один раз
- Теория
  - Декоратор использует только один уровень наследования,
  - все потомки переопределяют метод базового класса каждый по своему.
  - Наращивание функционала производится через инициализацию объекта базового класса вложенной конструкцией конструкторов потомков см. Пример.

- Пример. метод setType() переопределяется в потомках

[lesson\\_ch15/ch/ex38/access](#)

```
public class Basic {
    private String type;

    public Basic() { // конструктор нужен для Decorator конструктора
    }
    public Basic(String type) { // конструктор с предустановкой базового типа
        this.type = type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public String getType() {
        return type;
    }
}

• Декоратор
• public class Decorator extends Basic {
    protected Basic basic;

    public Decorator(Basic basic) { // добавили объект с реализацией Basic
        this.basic = basic;
    }
    public void setType(String s) { // делегируем в базовый класс
        basic.setType(s);
    }
    public String getType() { // делегируем в базовый класс
        return basic.getType();
    }
}
```

- Потомки декоратора

```

public class FoamCoffee extends Decorator {

    public FoamCoffee(Basic basic) {
        super(basic);
        setType(getType() + " & foam");
    }
}

public class StreamMilkCoffee extends Decorator {

    public StreamMilkCoffee(Basic basic) {
        super(basic);
        setType(getType() + " & streamed milk"); // переопределение метода
    }
}

public class WhipCreamCoffee extends Decorator {

    public WhipCreamCoffee(Basic basic) {
        super(basic);
        setType(getType() + " & whipped cream"); // переопределение базового класса
    }
}

```

- Применение. Вложенный вызов конструкторов потомков в тело предка

```

public static void app() {
    WhippedCream wp = new WhippedCream(new Basic());
    wp.setType("coffee "+wp.getType()); // ??? работает но не так как хотелось бы

    System.out.println(wp.getType()); // обязательно втащить coffee в
    конструктор
    // ВНИМАНИЕ именно вложенный вызов конструкторов реализует Mixing
    Decorator c = new FoamCoffee(new StreamedMilk(
        new WhippedCream(new Basic("coffee"))));
    System.out.println(c.getType());
}

```

- **ВНИМАНИЕ.** По моему совершенно дебильное решение

- неявный вызов методов декоратора
- скрытый механизм расширения функционала
- вызывается только при инициализации объекта, один раз

## Mixing и Dynamic Proxy

- Решение Mixing Proxy подмешивает методы лучше Decorator, но требует нисходящее преобразование
- Теория
  - создается объект Proxy с базой методов и объектов
  - нисходящее преобразование объекта Proxy до конкретного интерфейса позволяет вызвать метод
  - для вызова метода конкретного интерфейса требуется кастинг Mixing Proxy по этому интерфейсу
- Пример.
- Интерфейсы

[lesson\\_ch15/ch/ex39/local](#)

```
public interface IBasic {  
    void setString(String s);  
    String getString();  
}  
public interface ISerialNumber {  
    long getSerialNumber();  
}  
public interface ITimeStamp {  
    long getStamp();  
}
```

- Реализующие классы

```
public class Basic implements IBasic {  
    private String s;  
    @Override  
    public void setString(String s) {  
        this.s = s;  
    }  
    @Override  
    public String getString() {  
        return s;  
    }  
}  
public class TimeStamp implements ITimeStamp {  
    private final long timeStamp;  
    public TimeStamp() {  
        timeStamp = Calendar.getInstance().getTime().getTime();  
    }  
    @Override  
    public long getStamp() {  
        return timeStamp;  
    }  
}  
public class SerialNumber implements ISerialNumber {  
    private static long count = 1;  
    private final long sNumber = count++;  
  
    @Override  
    public long getSerialNumber() {  
        return sNumber;  
    }  
}
```

- Dynamic Mixing Proxy

```
• public class MixingProxy implements InvocationHandler {
    Map<String, Object> map;

    public MixingProxy(TwoTuple<Object, Class<?>>... pairs) { // array TwoTuple
        map = new HashMap<>();
        for (TwoTuple<Object, Class<?>> pair : pairs) {
            for (Method method : pair.second.getMethods()) { // из класса методы
                String name = method.getName(); // вытащили имя метода
                if (!map.containsKey(name)) { // если в базе нет метода
                    map.put(name, pair.first); // заполняем базу методов
                }
            }
        }
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    {
        String name = method.getName(); // вытащили строку метода
        Object delegated = map.get(name); // вытащили объект с реализацией
        метода
        return method.invoke(delegated, args); // идет подмена proxy
    }

    public static Object newInstance(TwoTuple... pairs) { // массив TwoTuple()
        Class[] interfaces = new Class[pairs.length]; // создали массив
        Class[]
        for (int i = 0; i < interfaces.length; i++) {
            interfaces[i] = (Class)pairs[i].second; // вытаскивается класс
        }
        // загрузчик класса по любому интерфейсу из группы
        ClassLoader cload = pairs[2].first.getClass().getClassLoader();
        return Proxy.newProxyInstance(
            cload,
            interfaces, // интерфейсы группы
            new MixingProxy(pairs) // конструктор Proxy
        );
    }
}
```

- Применение

```
• public static void app() {
    Object mixing = MixingProxy.newInstance( // передать массив объектов и классов
        new TwoTuple<Object, Class<?>>(new Basic(), IBasic.class),
        new TwoTuple<Object, Class<?>>(new TimeStamp(), ITimeStamp.class),
        new TwoTuple<Object, Class<?>>(new SerialNumber(), ISerialNumber.class)
    );
    IBasic b = (IBasic)mixing; // Object может быть любым
    ITimeStamp t = (ITimeStamp)mixing; //
    ISerialNumber s = (ISerialNumber)mixing; //
    b.setString("String Base");
    System.out.println("b:"+b.getString());
    System.out.println("t:"+t.getStamp());
    System.out.println("s:"+s.getSerialNumber());
}
```

## Latent и Generic

- Теория Latent типизация
  - как я понял это вызов метода у разных классов с разными интерфейсами, главное что метод был
  - если скажем есть метод speak() у разных классов, должна быть возможность его вызвать
  - Latent() работает так, чтобы вызов происходил автоматом из под объекта <T> то есть из <Object>
- Проблема Latent и Generic латентная типизация не поддерживается в Java
- Решение Latent через Reflection [lesson\\_ch15/ch/ex40/include](#)
- **ВНИМАНИЕ.** У Reflection очень низкая скорость это надо учитывать
- 
- Пример. Реализация Generic and Bounds, стандартный подход, без Latent [lesson\\_ch15/ch/ex40/local](#)
- Пример. Реализация Generic and Bounds, Latent через Reflection [lesson\\_ch15/ch/ex40/include](#)
- вызов метода происходит из под объекта <Object>

```
public static void perform(Object p) {
    Class<?> pClass = p.getClass();
    try {
        try {
            Method mSpeak = pClass.getMethod("speak");
            mSpeak.invoke(p); // вызов метода
        } catch (NoSuchMethodException e) {
            System.out.println(pClass.getSimpleName() + ": can't call speak()");
        }
        try {
            Method mSit = pClass.getMethod("sit");
            mSit.invoke(p); // вызов метода
        } catch (NoSuchMethodException e) {
            System.out.println(pClass.getSimpleName() + ": can't call sit()");
        }
    } catch (Exception e) { // all other exceptions
        throw new RuntimeException(pClass.getSimpleName(), e); // сообщение и
    }
}
```

- Пример. Реализация Latent через Reflection к последовательности объектов [lesson\\_ch15/ch/ex40/integra](#)
- **ВНИМАНИЕ.** Ключевой момент, это использование интерфейса Iterable<T>
- Пример. Реализация Latent через Reflection к Pets [lesson\\_ch15/ch/ex40/access](#)

```
public class Apply {
    public static <T, S extends Iterable<? extends T>> void apply(S list, Method f,
Object...args) {
        try {
            for (T t : list) { // итератор, так как создали ArrayList
                f.invoke(t, args); // вызвать метод с набором
            } // аргументов для данного объекта
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
public static void app() {
    try {
        Apply.apply(Pets.getList(10), Pet.class.getMethod("speak"));
    } catch (Exception e) {
        throw new RuntimeException();
    }
}
```

## Latent and Sequence Objects

- Проблема Latent and Sequence Objects чтобы работало Latent нужен универсальный метод fill()
  - метод должен работать с любыми последовательностями
  - нет универсального интерфейса Addable
- Решение Создать Adapter одного из 3х типов
- **ВНИМАНИЕ.** Универсальное применение метода fill() решается 3 способами
  - Adapter <IAddable> Composition, Adapter <IAddable> Inheritance, Adapter <IGenerator>

[lesson\\_ch15/ch/ex41/local](#)

## Адаптеры

- Адаптеры
  - добавление интерфейсов адаптеров позволяет методу fill() работать с любым классом
  - Адаптер Тип 1: класс хранения обворачивается классом оберткой с интерфейсом Addable<T>
  - Адаптер Тип 2: класс хранения наследуется классом оберткой с интерфейсом Addable<T>
- Пример. [lesson\\_ch15/ch/ex41/include](#)
- Адаптер Тип 1 [lesson\\_ch15/ch/ex41/access](#)

```
public class AdapterC<T> implements IAddable<T> { //класс адаптер для любой коллекции
    private Collection<T> c; // любой класс хранения, например SimpleQueue
    public AdapterC(Collection<T> c) { // передается по ссылке доступ есть снаружи
        this.c = c;
    }
    public void add(T item) {
        c.add(item);
    }
}
public class AdapterQ<T> implements IAddable<T> {
    private SimpleQueue<T> q;
    public AdapterQ(SimpleQueue<T> q) {
        this.q = q;
    }
    public void add(T item) {
        q.add(item);
    }
}
```

- Адаптер Тип 2. наследование класса хранения, но не всегда работает, с Collection конфликтует
- Генератор Адаптеров, работает с созданной коллекцией, автоматически определяет класс хранения<T>

```
public class Adapter {
    public static <T> AdapterC<T> getAutoAdapterC(Collection<T> c) {
        return new AdapterC<T>(c);
    }
    public static <T> AdapterQ<T> getAutoAdapterQ(SimpleQueue<T> q) {
        return new AdapterQ<T>(q);
    }
}
```

- Метод заполнения

```
public class Fill2 {
    public static <T> void fill(IAddable<T> s, Class<?> type, int size) {
        try {
            for (int i = 0; i < size; i++) {
                s.add(type.newInstance()); // добавить
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

- Применение.

```
public static void app() {
    List<Contract> clist = new ArrayList<>(); // втаскиваем коллекцию через адаптер,
    Fill12.fill(new AdapterC<>(clist), Contract.class, 10); // который даст метод

    SimpleQueue<Contract> cqueue = new SimpleQueue<>();
    Fill12.fill(new AdapterQ<>(cqueue), Contract.class, 10); // адаптер тип 1

    ArrayList<Contract> dlist = new ArrayList<>(); // применен генератор адаптера
    Fill12.fill(Adapter.getAutoAdapterC(dlist), Contract.class, 10); // адаптер тип 1
    System.out.println("dlist :" + dlist);

    SimpleQueueAdd<Contract> dqueue = new SimpleQueueAdd<>(); // адаптер тип 2
    Fill12.fill(dqueue, Contract.class, 10); // втаскиваем коллекцию через адаптер
}
```

## Latent и Универсальные Адаптеры

- Универсальные Адаптеры [lesson\\_ch15/ch/ex42/local](#)
  - созданы для получения обобщенного кода
  - используется Pattern «Strategy» и объекты функции
- Объект функция
  - это объекты которые выглядят как метод, работают как метод
  - в отличие от методов, могут передаваться при вызовах, хранить состояние между вызовами
- **ВНИМАНИЕ.** Данный метод очень мощный и с Pattern Strategy обязательно разобраться
- **Теория.** В общем все довольно сложно и надо будет разобраться Pattern Strategy ????
- Пример. реализация математики на разных классах Number [lesson\\_ch15/ch/ex42/local](#)
- Пример. реализация на двух разных Custom классах [lesson\\_ch15/ch/ex42/access](#)

## Arrays Массивы

- Arrays Массивы
  - линейная последовательность элементов одного типа, объекты или примитивные типы
  - произвольный доступ, высокая скорость доступа
  - фиксированный размер
- **ВНИМАНИЕ.** автоматическая упаковка работает поэлементно, но НЕ работает для всего массива
- Нельзя назначить массиву Integer ссылку на массив int >> Integer[] = int[]
- ArrayList
  - коллекция как замещение массива, может хранить только объекты
  - произвольный доступ, доступ работает медленее чем массивы
  - произвольный динамически изменяемый размер
- Инициализация массивов
  - массивы примитивных типов по умолчанию заполняются согласно типа объекта
  - массивы объектов по умолчанию заполняются <null>
- Инициализация массивов способы
  - агрегатная инициализация, то есть инициализация { }
  - динамическая агрегатная инициализация new Array[] { }
- **ВНИМАНИЕ.** Избыточная инициализация если при объявлении динамическая агрегатная инициализация
- Пример. Нормальная агрегатная или требуемая динамическая инициализация

```
// необходимая агрегатная инициализация
BerylliumSphere[] d = {           // агрегатная инициализация
    new BerylliumSphere(), new BerylliumSphere()
};

// необходимая динамическая агрегатная инициализация
BerylliumSphere [] e;           // объявление и инициализация разделены
e = new BerylliumSphere[]{      // создание массива
    new BerylliumSphere(), new BerylliumSphere(),
};
check(new BerylliumSphere[]{new BerylliumSphere()}); // метод иначе не
работает
```
- Пример. Избыточная
- // избыточная динамическая агрегатная инициализация

```
BerylliumSphere[] f = new BerylliumSphere[] {
    new BerylliumSphere(), new BerylliumSphere(),
    new BerylliumSphere()
};
```

## Возврат массива

- Возврат массива в Java возврат массива это стандартная процедура

[lesson\\_ch16/ch/ex2/local](#)

## Многомерные массивы

- Многомерные массивы
  - все измерения могут быть разной длины
  - **ВНИМАНИЕ.** печать многомерным массивов Arrays.deepToString()
- Инициализация массивов примитивов

```
public static void app() {  
    int [][] a = { {1,2,3}, {4,5,6} };  
    int [[[ ] b = new int[2][2][4];  
    System.out.println(Arrays.deepToString(a));  
    System.out.println(Arrays.deepToString(b));  
}
```
- Инициализация массивов с объектами
  - автоупаковка НЕ работает с массивами в целом нельзя назначить Integer[] = int[]

[lesson\\_ch16/ch/ex3/local](#)

[lesson\\_ch16/ch/ex3/include](#)

## Array and Generic

- Array and Generic      массивы и обобщения плохо работают друг с другом [lesson\\_ch16/ch/ex8/local](#)
  - создать параметризованный массив Peel<Banana>[] peels; с принудительным cast type
  - можно создать ссылку на параметризованный массив
  - параметризованный массив объектов все равно сможет записать данные в Object[]
- **ВНИМАНИЕ.** Массив объектов созданный с <T> типом все равно может работать с Object[]
- Реально создать
  - работает. ссылку на параметризованный массив List<String>[] с явным типом
  - List<String>[] ls; // ссылка на массив параметризованных списков
  - работает. ссылку на параметризованный массив List[] с типом <T> Generic и кастингом

```
public List<T>[] lists;  
this.lists = (List<T>[])new List[10]; // casting
```
  - работает. ссылку на массив с типом <T> Generic и последующим кастингом

```
public T[] objects;  
this.objects = (T[])new Object[10]; // casting
```
  - работает. массив List<String>[] с явным типом

```
List<BerylliumSphere>[] cp = (List<BerylliumSphere>[]) new List[10];  
//параметризованный массив создан
```
  - работает. массив List<T>[] с типом Generic
  - **public List<T>[] clists = (List<T>[])new List[10];** //создание массива с Generic
- Создать не получится
  - не работает. массив T[] с типом Generic
  - **public T[] cobjects = (T[])new Object[10];** //
- Работать не будет [lesson\\_ch16/ch/ex8/local](#)
  - создание массива с <T> Generic типом созданный с кастингом от Object[]
  - возникает конфликт интересов компилятора и Runtime
  - компилятор думает что это массив <T>[] заставляет работать с ним как с T[] и не принимает Object
  - из за стирания во время RunTime массив превращается в Object[] и принимает только Object
- **ВНИМАНИЕ.** в результате массив создается но работать с ним невозможно

```
public T[] cobjects = (T[])new Object[10]; // не работает
```
- Пример. [lesson\\_ch16/ch/ex9/access /ex10/access](#)

```
public class PeelMethod {  
    public static<T> T[] get() {  
        T[] array = (T[]) new Object[5];  
        return (T[])array;  
    }  
}
```

// не работает, то есть да массив дает  
//но он затирается и с ним  
// невозможно работать

## Создание тестовых данных

- Заполнение массивов методом Arrays.fill()
  - массивы примитивово заполняются одним значением

[lesson\\_ch16/ch/ex11/local](#)

## Генераторы данных

- Генератор данных это класс использует интерфейс IGenerator
- Пример. Генератор последовательных данных

```
public class GenSeq {  
    public static class GenChar implements IGenerator<Character> {  
        private static char[] chs = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();  
        private int index = -1;  
        public Character next() {  
            index++;  
            return new Character(chs[index%chs.length]); // остаток длины массива  
        }  
    }  
    public static class GenInt implements IGenerator<Integer> {  
        private int value = 0;  
        public Integer next() {  
            return value++;  
        }  
    }  
}
```

[lesson\\_ch16/ch/ex11/include](#)

- Пример. Генератор случайных данных, тоже самое только с Random()

```
public class GenRnd {  
    private static Random rnd = new Random();  
    public static class GenChar implements IGenerator<Character> {  
        private static char[] chs = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();  
        public Character next() {  
            return new Character(chs[rnd.nextInt(chs.length)]); // случайный индекс  
        }  
    }  
    public static class GenInt implements IGenerator<Integer> {  
        private int mod = 10000;  
        public Integer next() {  
            return rnd.nextInt(mod); // диапазон  
        }  
    }  
}
```

- Применение. Класс тестирования генераторов, используется Reflection

```
public class Include {  
    public static void genTest(Class<?> type) {  
        int size = 10; // длина данных  
        for (Class<?> aClass : type.getClasses()) { // получить внутренние классы  
            System.out.printf("%-10s:", aClass.getSimpleName());  
            try {  
                IGenerator<?> gen = (IGenerator<?>)aClass.newInstance();  
                for (int i = 0; i < size; i++) {  
                    System.out.print(gen.next() + " ");  
                }  
            } catch (Exception e) {}  
            System.out.println();  
        }  
    }  
    public static void app() {  
        genTest(GenSeq.class); // прокачать генераторы всех внутренних классов  
    }  
}
```

```
        genTest(GenRnd.class); // прокачать генераторы Random внутренних классов
    }
```

## Применение генераторов для создания массивов

- Создание массивов с помощью генераторов производится в два этапа
- первый генератор создает массив объектов соответствующих примитивам
  - себя метод заполнения массивов, и метод создания массивов на базе Array.newInstance()
- второй конвертер примитивов конвертирует массив объектов в массив примитивов (простое заполнение)
- Пример. Генератор массивов
  - класс GenList создан на базе ArrayList и его методов add() и toArray(arg)
  - метод add() заполняет ArrayList, метод toArray(arg) создает массив[] из ArrayList
- **ВНИМАНИЕ. ОБЯЗАТЕЛЬНО** в метод ArrayList.toArray(arg) заполнять arg чтобы избежать стирания

```
public class GenList<T> extends ArrayList<T> {
    public GenList(IGenerator<T> gen, int size) {
        for (int i = 0; i < size; i++) {
            add(gen.next()); // add to ArrayList
        }
    }
    public static <T> GenList<T> get(IGenerator<T> gen, int size) {
        return new GenList<T>(gen, size);
    }
}
public class Gen { // Array IGenerator
    public static <T> T[] getArray(T[] array, IGenerator<T> gen) {
        return new GenList<T>(gen, array.length).toArray(array);
    } // ОБЯЗАТЕЛЬНО Экземпляр в toArray
    public static <T> T[] getArray(Class<?> type, IGenerator<T> gen, int size) {
        T[] array = (T[]) Array.newInstance(type, size); // массив из класса
        return new GenList<T>(gen, array.length).toArray(array); // заполнить массив
    }
}
```

- Пример. Конвертер в массив примитивов

[lesson\\_ch16/ch/ex11/integra](#)

```
public class ConvTo {
    public static char[] primitive(Character[] array) {
        char[] res = new char[array.length];
        for (int i = 0; i < array.length; i++) {
            res[i] = array[i]; // автоматическая распаковка
        }
        return res;
    }
    public static int[] primitive(Integer[] array) {
        int[] res = new int[array.length];
        for (int i = 0; i < array.length; i++) {
            res[i] = array[i]; // автоматическая распаковка
        }
        return res;
    }
}
```

- Применение

[lesson\\_ch16/ch/ex11/integra](#)

```
public static void app() {
    Integer[] array = {9, 8, 7, 6};
    Integer[] bArray = Gen.getArray(array, new GenSeq.GenInt()); // заполнить Integer
    Integer[] dArray = Gen.getArray(Integer.class, new GenSeq.GenInt(), 15); //
    Integer[] fArrayRnd = Gen.getArray(array, new GenRnd.GenInt());
    int[] aInts = ConvTo.primitive(fArrayRnd); // случайный массив заполнить
    int[] bInts = ConvTo.primitive(Gen.getArray(Integer.class, new GenRnd.GenInt(), 10));
    char[] chs = ConvTo.primitive(Gen.getArray(Character.class, new GenRnd.GenChar(), 10));
}
```

## Класс Arrays

- Класс Arrays содержит 6 статических методов работы с массивами
  - equals(), deepEquals()
  - fill()
  - sort()
  - binarySearch()
  - toString(), deepToString()
  - hashCode()
  - asList()

## Копирование массивов

- Копирование массива System.arraycopy()
  - это очень быстрый метод копирования массивов
- **ВНИМАНИЕ.** При копировании массивов объектов, копируются только ссылки, объекты те же самые
- Пример. [lesson\\_ch16/ch/ex18/access](#)

## Сравнение массивов

- Сравнение массивов производит Arrays.equals()
  - работает для всех объектов и примитивных типов
  - сравнение идет по числу элементов массивов и для каждого элемента вызывается Class.equals()
- **ВНИМАНИЕ.** Для String практически всегда Object.equals() поэтому сравнение идет по содержимому
- Очень редко требуется String == String полное равенство что это одна и та же ссылка
- **ВНИМАНИЕ.** Для MyClass чтобы сделать сравнение по конкретному полю надо добавить свое equals()
- Пример. [lesson\\_ch16/ch/ex19/access](#)

## Сравнение элементов массивов

- Сравнение элементов массивов
  - при сравнении «отделяется изменяющийся код от постоянного»
  - используется паттерн «Стратегия», «изменяющийся код» заключается в класс «объект Стратегии»
- Два способа сравнения объектов через интерфейсы Comparable<T> и Comparator<T>
- Интерфейс Comparable <T> уровень класс пользователя
  - в классе реализуется метод CompareTo
  - сортировка автоматом через Arrays.sort(array)
- Интерфейс Comparator <T> уровень внутренний класс Comparator
  - в классе создается подкласс Comp с интерфейсом Comparator<T>
  - сортировка через явное указание на объект компаратор Arrays.sort(array, new Comp())
- Пример. Пример комбинированной сортировки [lesson\\_ch16/ch/ex21/access](#)

- Пример. сортировка forward Comparable <T> и reverse Comparator<T> [lesson\\_ch16/ch/ex21/access](#)

```

public class BerylliumSphere implements Comparable<BerylliumSphere> {
    private static long counter;
    private final long id = counter++;
    // Generator
    public static IGenerator<BerylliumSphere> gen() {
        return new IGenerator<BerylliumSphere>() {
            @Override
            public BerylliumSphere next() {
                return new BerylliumSphere();
            }
        };
    }
    // Comparator
    public static Comparator<BerylliumSphere> getComp() {
        return new Comparator<BerylliumSphere>() {
            @Override
            public int compare(BerylliumSphere o1, BerylliumSphere o2) {
                if (o1.id > o2.id) {
                    return -1;
                }
                if (o1.id < o2.id) {
                    return 1;
                }
                return 0;
            }
        };
    }
    // Comparable
    @Override
    public int compareTo(BerylliumSphere o) {
        if (id < o.id) {
            return -1;
        }
        if (id > o.id) {
            return 1;
        }
        return 0;
    }
    @Override
    public String toString() {
        return "Sphere " + id;
    }
}

```

- Применение.

- ```

public static void app() {
    Random rnd = new Random();
    BerylliumSphere[] baseArray = Gen.getArray(BerylliumSphere.class,
  BerylliumSphere.gen(), 25);
    BerylliumSphere[] bArray = new BerylliumSphere[7];
    for (int i = 0; i < bArray.length; i++) {
        int index = rnd.nextInt(baseArray.length);
        bArray[i] = baseArray[index];
    }
    System.out.println("unsorted:" + Arrays.toString(bArray));
    Arrays.sort(bArray);
    System.out.println("forward :" + Arrays.toString(bArray));
    Arrays.sort(bArray, BerylliumSphere.getComp());
    System.out.println("reverse :" + Arrays.toString(bArray));
}
```

## Сортировка массива

- Сортировка массива
- Пример. Сортировка массива String[] с помощью Arrays.sort()

[lesson\\_ch16/ch/ex22/local](#)

## Поиск в сортированном массиве

- Поиск в сортированном массиве
  - Arrays.binarySearch() используется для поиска в сортированном массиве
- **ВНИМАНИЕ.** если в массиве есть дубликаты, будет найден любой первый попавшийся
- если сортировка была с Comparator<> то и binarySearch() надо использовать с Comparator<>
- **ВНИМАНИЕ.** для отсечения дубликатов и автосортировки применять TreeSet()
- для хранения в порядке вставки без дубликатов LinkedHashSet()
- Пример.

```
public static void app() {  
    GenRnd.GenInt cg = new GenRnd.GenInt();  
    int [] aInt = ConvTo.primitive(Gen.getArray(new Integer[25],cg)); // сгенерили  
    Arrays.sort(aInt); // сортировка  
    while (true) {  
        int rndVal = cg.next(); // следующее Int  
        int pos = Arrays.binarySearch(aInt,rndVal);  
        if (pos >= 0) {  
            System.out.println("Location of "+rndVal+ " is "+ pos+  
                " , aInt["+pos+"] = "+aInt[pos]);  
            break;  
        }  
    }  
}
```

## ВЫВОДЫ

- **ВНИМАНИЕ.** Следует отдавать предпочтение контейнерам вместо массивов
- Массивы использовать в крайнем случае, когда нужна производительность и точно известно, что причина низкой производительности из за неиспользования массива
- Массивы примитивов имеют хорошую поддержку в языке.
- При любой возможности заменять их контейнерами

# Containers Контейнеры

## Containers Контейнеры

- Containers Контейнеры
  - контейнеры это структуры Collections List, Set и Map
- Заполнение контейнеров
  - Collections.nCopy(size,new Object())
  - Collections.fill( Collectioon<> col, new Objects)

[lesson\\_ch17/ch/ex01/local](#)

создать коллекцию несколько копий объекта  
заполнить коллекцию объектом

## Генераторы List<T> Set<T>

- Генераторы List<T> Set<T>
  - все классы Collection имеют конструктор с параметром Collection<>
  - можно в качестве аргумента подсунуть ArrayList<> с генератором компонентов заданного типа
- Пример.  
[lesson\\_ch17/ch/ex01/include](#)
- Класс расширение ArrayList<T>
- ```
public class CData<T> extends ArrayList<T> {
    public CData(IGenerator<T> gen, int size) {
        for (int i = 0; i < size ; i++) {
            add(gen.next());
        }
    }
    public static<T> CData<T> getList(IGenerator<T> gen, int size ) {
        return new CData<T>(gen,size);
    }
}
```

// генератор списка реализация внешняя  
// заполнение стандартным методом

// получить готовый список методом
- Класс генератор IGenerator<String>
- ```
public class GenGov implements IGenerator<String> { // реализация генератора строк
    private String[] strings = ("strange women lying in ponds " +
        "distributing swords is no basis for a system of " +
        "government").split(" ");
    private int index;
    @Override
    public String next() {
        return strings[index++ % strings.length]; // не превышая размер массива
    }
}
```
- Применение. Генерация Set, List
- ```
public static void app() {
    System.out.println("Initialization Set with IGenerator:");
    List<String> listGen = new CData<>(new GenGov(), 25); // генератор и список
    Set<String> lhset = new LinkedHashSet<>( listGen);
    System.out.println(listGen);
    System.out.println(lhset);
    System.out.println("Additional Method:");
    lhset.addAll(new CData<>(new GenGov(), 15));
    System.out.println(lhset);

    System.out.println("\nArrayList<> Generator Random String:");
    ArrayList<String> aList = CData.getList(new GenRnd.GenStr(9), 10);
    System.out.println(aList);

    System.out.println("\nHashSet<> Generator Random Integer:");
    Set<Integer> hset = new HashSet<>(CData.getList(new GenRnd.GenInt(),10));
    System.out.println(hset);

    System.out.println("\nTreeSet<> Generator Random Integer:");
    Set<Integer> tset = new TreeSet<>(CData.getList(new GenRnd.GenInt(),10));
    System.out.println(tset);
}
```

## Генераторы Map

- Генераторы Map
  - все классы Collection имеют конструктор с параметром Collection<>
  - поэтому можно в качестве аргумента подсунуть ArrayList<> с генератором
  - для Map дополнительно используется класс Pair с парой классов Collection<> Key, Value
  - используется два генератора IGenerator<T> или комбинация с Iterable<T> для генерации данных

### Пример.

[lesson\\_ch17/ch/ex01/integra](#)

- Используется мощный класс MapData
- используется класс с интерфейсами IGenerator<Pair> и Iterable<Integer>
- ```
public class Pair<K,V> { // Class for Map IGenerator
    public final K key; // for read only
    public final V value; // for read only

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}

public class MapData<K, V> extends LinkedHashMap<K, V> { // конструктор
    public MapData(IGenerator<Pair<K, V>> gen, int size) { // ген ключей, значений
        for (int i = 0; i < size; i++) {
            Pair<K, V> p = gen.next(); // сгенерили объект Pair() генератор внешний
            put(p.key, p.value); // 
        }
    }

    public MapData(IGenerator<K> genK, IGenerator<V> genV, int size) { // два генератора
        for (int i = 0; i < size; i++) { // генератор ключей, значений
            put(genK.next(), genV.next()); // два разных генератора
        }
    }

    public MapData(IGenerator<K> genK, V value, int size) { // генератор ключей
        for (int i = 0; i < size; i++) {
            put(genK.next(), value); // генератор ключей и значение
        }
    }

    public MapData(Iterable<K> genK, IGenerator<V> genV) { // итератор и генератор
        for (K key : genK) {
            put(key, genV.next()); // два разных генератора
        }
    }

    public MapData(Iterable<K> genK, V value) { // итератор и значение
        for (K key : genK) {
            put(key, value); // генератор ключей и значение
        }
    }
}
```

### Применение

[lesson\\_ch17/ch/ex01/integra](#)

- ```
public static void app() {
    // MapData IGenerator<Pairs<K,V>>
    Map<Integer, String> aMap = new MapData<Integer, String>(new Letters(), 11);
    // MapData IGenerator<Char>, IGenerator<String>:"");
    Map<Character, String> bMap = new MapData<>(new GenSeq.GenChar(),
        new GenRnd.GenStr(3), 11);
    // MapData IGenerator<Char>, Value<String>:"");
    Map<Character, String> cMap = new MapData<>(new GenSeq.GenChar(), "Value", 6);
    // MapData Iterable<Integer>, IGenerator<String>:"");
    Map<Integer, String> dMap = new MapData<>(new Letters(), new GenRnd.GenStr(3));
    // MapData Iterable<Integer>, Value<String>:"");
    Map<Integer, String> eMap = new MapData<>(new Letters(), "Value"); // итератор
}
```

## Использование классов Abstract

- Реализация генератора на базе классов AbstractMap, AbstractSet      применение шаблона «Легковес»
  - создание генераторов тестовых данных на базе Abstract Collection или Map Classes
  - сначала наследуется Abstract Collection или Map класс, затем реализуется его метод iterator()
- Пример.
  - класс FlyWeightMap наследует AbstractMap <> и реализует метод entrySet()
  - класс EntrySet()      наследует AbstractSet<> реализует методы size(), iterator(), создает entries
  - класс Entry()      наследует Map.Entry<>    реализует методы getKey(), getValue(), equals()
- ```
public class Countries {  
    private CMap cMap = new CMap();  
    public static Map<String, String> map = new FlyWeightMap(); // карта AbstractMap  
    public static List<String> names = new ArrayList<>(map.keySet()); // имена стран  
    private static class FlyWeightMap extends AbstractMap<String, String> {  
        private static Set<Map.Entry<String, String>> entries =  
            new EntrySet(CMap.DATA.length); // полный комплект  
        private static class Entry implements Map.Entry<String, String> { // класс Entry  
            int index;  
            public Entry(int index) { // конструктор создается с значением index  
                this.index = index;  
            }  
            public String getKey() {  
                return CMap.DATA[index][0];  
            }  
            public String getValue() {  
                return CMap.DATA[index][1];  
            }  
            public String setValue(String value) {  
                throw new RuntimeException(); // CMap.DATA final no write  
            }  
            public boolean equals(Object o) {  
                return CMap.DATA[index][0].equals(o);  
            }  
            public int hashCode() {  
                return CMap.DATA[index][0].hashCode();  
            }  
        }  
        static class EntrySet extends AbstractSet<Map.Entry<String, String>> {  
            private int size;  
            public EntrySet(int size) { // конструктор EntrySet заполняет длину  
                if (size < 0) this.size = 0;  
                if (size > CMap.DATA.length) this.size = CMap.DATA.length;  
                else this.size = size;  
            }  
            public int size() //AbstractSet >> Abstract Collection >> implementation  
            return size;  
            }  
            public Iterator<Map.Entry<String, String>> iterator() { // Iterator method  
                return new Iterator<Map.Entry<String, String>>() {  
                    private Entry entry = new Entry(-1); // в общем что то создается  
                    public boolean hasNext() {  
                        return entry.index < (size - 1);  
                    }  
                    public Map.Entry<String, String> next() {  
                        entry.index++;  
                        return entry;  
                    }  
                };  
            } // EntrySet.iterator()  
        } //EntrySet Class  
        public Set<Map.Entry<String, String>> entrySet() // AbstractMap implementation  
        return entries; // << EntrySet << Entry  
    }  
} // FlyWeightMap Class
```

```

// Static Methods
    public static Map<String, String> select(final int size) {
        return new FlyWeightMap() {
            public Set<Map.Entry<String, String>> entrySet() { // переопределение
                return new EntrySet(size);
            }
        };
    }
    public static Map<String, String> capitals() {
        return map; // полная карта
    }
    public static Map<String, String> capitals(int size) {
        return select(size); // частичная карта
    }
    public static List<String> names() {
        return names; // полный список стран
    }
    public static List<String> names(int size) {
        return new ArrayList<>(select(size).keySet()); // частичная карта
    }
}

```

- Применение
- `public static void app() {`  
 `System.out.println("Map with some countries");`  
 `System.out.println(Countries.capitals(5));`  
 `System.out.println(Countries.names(5));`  
 `System.out.println("\nMap Generator for Map");`  
 `Map<String, String> hMap = new HashMap(Countries.capitals(3));`  
 `System.out.println("\nMap Generator for Set");`  
 `Set<String> hSet = new HashSet<>(Countries.names(3));`  
 `System.out.println("\nMap Generator for List");`  
 `List<String> aList = new ArrayList<>(Countries.names(5));`  
 `System.out.println(Countries.capitals().get("BRAZIL"));`  
`}`

## Реализация генераторов на базе AbstractList

- Реализация генераторов на базе AbstractList
  - класс CIntList наследует класс AbstractList и реализует методы `get()` и `size()`
  - конструктор создает список нужного размера
  - метод `get()` как раз работает как генератор
- Пример.
 

[lesson\\_ch17/ch/ex01/thrown](#)
- `public class CIntList extends AbstractList<Integer>{`  
 `private int size;`  
 `private int coeff[] = {1, 5, 7, 9, 15};`  
 `public CIntList(int size) {`  
 `if(size < 0) this.size = 0;`  
 `else this.size = size;`  
 `}`  
 `@Override`  
 `public Integer get(int index) {`  
 `return Integer.valueOf(index * coeff[index %coeff.length]); // генератор`  
 `}`  
 `@Override`  
 `public int size() {`  
 `return size;`  
 `}`  
`}`
- Применение
- `public static void app() {`  
 `CIntList cList = new CIntList(25);`  
 `System.out.println(cList);`  
`}`

## Реализация генераторов на базе AbstractMap

- Реализация генераторов на базе AbstractMap, LinkedHashSet
  - в отличие от предыдущего способа, здесь EntrySet на базе LinkedHashSet
- Последовательность реализации
- создать класс расширяющий AbstractMap<K,V>
  - создать поле размера int size
  - создать конструктор, который заполняет size
- создать класс Entry с интерфейсом Map.Entry<K,V>
  - создать поле int index
  - создать конструктор Entry(index) то есть при создании будет заполняться index
  - реализовать метод getKey() >> index
  - реализовать метод getValue() >> честный генератор с любой реализацией
  - реализовать метод setValue() >> выдает RunTimeException() так как генератор только на чтение
- реализовать метод Map.Entry<K,V> entrySet()      пока в виде заглушки
  - создать Set<Map.Entry<K,V> на базе LinkedHashSet() >> entries
  - прогнать по размеру size и добавить все new Entry()
  - вернуть entries

### Пример.

[lesson\\_ch17/ch/ex01/thrown](#)

```
public class CIntMap extends AbstractMap<Integer, String> {  
    private int size;  
    private static String[] strings = "A B C D E F G H I J K L M N O P Q R".split(" ");  
    public CIntMap(int size) {  
        if (size < 0) {  
            this.size = 0; // сделать размер = 0  
        } else {  
            this.size = size; // нормальный размер  
        }  
    }  
    private static class Entry implements Map.Entry<Integer, String> {  
        int index;  
        public Entry(int index) {  
            this.index = index;  
        }  
        public Integer getKey() {  
            return index;  
        }  
        public String getValue() {  
            String s = strings[index%strings.length] + (index/strings.length); //01,H5  
            return s;  
        }  
        public String setValue(String value) {  
            throw new RuntimeException();  
        }  
    }  
    public Set<Map.Entry<Integer, String>> entrySet() {  
        Set<Map.Entry<Integer, String>> entries = new LinkedHashSet<>();  
        for (int i = 0; i < size; i++) {  
            entries.add(new Entry(i)); // добавляет Entry() объект  
        }  
        return entries;  
    }  
}
```

### Применение.

[lesson\\_ch17/ch/ex01/thrown](#)

```
public static void app() {  
    System.out.println("List Generator based on AbstractList:");  
    CIntMap cMap = new CIntMap(25);  
    System.out.println(cMap);  
}
```

## Реализация генераторов на базе AbstractMap полная версия

- Реализация генераторов на базе AbstractMap, AbstractSet lesson\_ch17/ch/ex05/access
- Последовательность реализации
- создать класс расширяющий AbstractMap<K,V>
  - создать поле размера int size
  - создать конструктор, который заполняет size
- создать класс Entry с интерфейсом Map.Entry<K,V>
  - создать поле int index
  - создать конструктор Entry(index) то есть при создании будет заполняться index
  - реализовать метод getKey() >> index
  - реализовать метод getValue() >> честный генератор с любой реализацией
  - реализовать метод setValue() >> выдает RunTimeException() так как генератор только на чтение
- создать класс EntrySet
  - создать поле size
  - создать конструктор и пробросить в него size это основа для итератора
  - реализовать метод EntrySet.iterator() на базе внутреннего класса Iterator{}
  - реализовать внутренний EntrySet.класс Iterator{}
  - для класса Iterator создать поле index, по которому будет перемещаться Iterator
  - для класса Iterator реализовать метод hasNext() возвращает (index< size)
  - для класса Iterator реализовать метод next() ГЛАВНОЕ создавать объект newEntry(index) на лету
  - реализовать метод EntrySet.size() возвращает просто size
- реализовать метод Map.Entry<K,V> entrySet()
  - возвращает объект newEntrySet(size)

- Пример. на базе Map<Integer,String> lesson\_ch17/ch/ex05/access
- ```
public class CIntMap extends AbstractMap<Integer, String> {  
    private int size;  
    private static String[] strings = "A B C D E F G H I J K L M N O P Q R".split(" ");  
    public CIntMap(int size) {  
        if (size < 0) {  
            this.size = 0; // сделать размер = 0  
        } else {  
            this.size = size; // нормальный размер  
        }  
    }  
    private static class Entry implements Map.Entry<Integer, String> {  
        int index;  
        public Entry(int index) {  
            this.index = index;  
        }  
        public Integer getKey() {  
            return index;  
        }  
        public String getValue() {  
            String s = strings[index%strings.length]+ (index/strings.length); // 01,H5  
            return s;  
        }  
        @Override  
        public String setValue(String value) {  
            throw new RuntimeException();  
        }  
    }  
}
```

- ```

private static class EntrySet extends AbstractSet<Map.Entry<Integer, String>> {
    private int size;
    public EntrySet(int size) {
        this.size = size;
    }
    @Override
    public Iterator<Map.Entry<Integer, String>> iterator() {
        return new Iterator<Map.Entry<Integer, String>>() {
            private int index = 0;
            @Override
            public boolean hasNext() {
                return index < size;
            }

            @Override
            public Map.Entry<Integer, String> next() { // создавать объект по ходу
                index++;
                return new Entry(index);
            }
        };
    }
    @Override
    public int size() {
        return size;
    }
}
@Override
public Set<Map.Entry<Integer, String>> entrySet() {
    return new EntrySet(size);
}
}

```

- Применение.

- ```

public static void app() {
    Map<Integer, String> cMap = new CIntMap(25);
    System.out.println(cMap);

    Set<Map.Entry<Integer, String>> entries = cMap.entrySet();
    Iterator<Map.Entry<Integer, String>> it = entries.iterator();
    for (int i = 0; i < entries.size(); i++) {
        Map.Entry entry = it.next();
        System.out.print(entry.getKey() + "=" + entry.getValue() + " ");
    }
    System.out.println();
    for (Map.Entry<Integer, String> entry : entries) {
        System.out.print(entry.getKey() + " " + entry.getValue() + " ");
    }
}

```

## Функциональность Collection

- Функциональность Collection

[lesson\\_ch17/ch/ex06/local](#)

Функция	Описание
boolean <b>add(T)</b>	добавить элемент
boolean <b>addAll(Collection&lt;?&gt;)</b>	добавить группу на базе Collection
void <b>clear()</b>	очистить
boolean <b>contains(T)</b>	проверить содержится ли элемент
boolean <b>containsAll(Collection&lt;?&gt;)</b>	проверить содержит ли группа элементов
boolean <b>isEmpty()</b>	пустая ли коллекция
Iterator <T> <b>iterator()</b>	возврат итератора
boolean <b>remove(T)</b>	удалить элемент
boolean <b>removeAll(T)</b>	удалить группу элементов
boolean <b>retainAll(T)</b>	удалить все кроме группы элементов
int <b>size()</b>	вернуть размер
Object[] <b>toArray()</b>	преобразовать в массив Object[]
<T> T[] <b>toArray(T[] a)</b>	тоже самое но работает с T[] препятствует Erasure

## Необязательные методы Collection

- Необязательные методы Collection
  - необязательные методы, это расширения интерфейсов
  - созданы чтобы снизить число интерфейсов и оставить возможности в пакете

## Неподдерживаемые операции Collection

- Неподдерживаемые операции Collection
  - возникают когда методы применяются к коллекциям фиксированного размера
- **ВНИМАНИЕ.** List<> созданный на базе Arrays.asList() создает Collection с НЕИЗМЕННЫМ размером массива
- **ВНИМАНИЕ.** Для создания изменяемой коллекции использовать new ArrayList(Arrays.asList())
  - возникают когда методы применяются к коллекциям которые запрещено изменять
- **ВНИМАНИЕ.** методы Collections.unmodifiable() фиксируют Collection<> полностью делают его константой
- Пример.

[lesson\\_ch17/ch/ex06/include](#)

[lesson\\_ch17/ch/ex06/include](#)

## Методы List

- Методы List
- Базовый функционал List
  - методы add(), get(), iterator()
- Расширенный функционал ArrayList
  - listIterator(), hasNext(), next(), hasPrevious(), previous()
  - set()
  - indexOf()
- Расширенный функционал LinkedList
  - addFirst(),
  - getFirst()
  - removeFirst()
  - removeLast()
- Пример. Самодельный связный список с итератором
  - мощный пример ручной реализации практически всех функций

[lesson\\_ch17/ch/ex07/local](#)

[lesson\\_ch17/ch/ex08/access](#)

## Set и порядок хранения

- Set и порядок хранения

Множество	Описание
Set <Interface>	базовый интерфейс множеств
HashSet	реализация с быстрым поиском реализация hashCode()
TreeSet	реализация с сортировкой реализация <Comparable>
LinkedHashSet	реализация аналогична HashSet хранение в порядке вставки, hashCode()

- **ВНИМАНИЕ.** если нет предпочтений, то лучше выбрать HashSet как наиболее производительный вариант

## Методы Set

- HashSet<> или LinkedHashSet<> [lesson\\_ch17/ch/ex09/local](#)

- метод equals() сравнение на повторность экземпляра
  - метод hashCode() уникальный код для каждого экземпляра

- **ВНИМАНИЕ.** Хороший стиль при переопределении equals() также переопределять hashCode()

- TreeSet<>

- метод equals() сравнение на повторность экземпляра
  - метод compareTo() сравнение компонентов, реализация <Comparable>

- **ВНИМАНИЕ.** отсутствие методов в объектах хранения делает невозможным их хранение в Set<>

- **ВНИМАНИЕ.** метод compareTo() должен реализовывать ИМЕННО СРАВНЕНИЕ, вычитание НЕ использовать

- Пример. объекты хранения HashType для HashSet<> и TreeType для TreeSet [lesson\\_ch17/ch/ex09/local](#)

```
• public class SetType { // базовый класс хранения содержит общие компоненты
    int i;
    public SetType(int i) {
        this.i = i;
    }
    public boolean equals(Object o) { // метод equals() общий для всех классов хранения
        if ((o instanceof SetType) && i == ((SetType) o).i) {
            return true;
        }
        return false;
    }
    @Override
    public String toString() {
        return Integer.toHexString(i);
    }
}
• public class HashType extends SetType{ // расширение SetType для HashSet<>
    public HashType(int i) {
        super(i);
    }
    public int hashCode() { // реализация для HashSet<>
        return i;
    }
}
• public class TreeType extends SetType implements Comparable<TreeType>{ // для TreeSet<>
    public TreeType(int i) {
        super(i);
    }
    @Override
    public int compareTo(TreeType arg) { // от предка SetType
        if (arg.i < i) return -1;
        if (arg.i > i) return 1;
        return 0;
    }
}
```

- Применение. Проверка работы множеств Set<> с типами хранения данных [lesson\\_ch17/ch/ex09/local](#)
- ```

public static void app() {
    TypeSets.checkHash(new HashSet());
    TypeSets.checkHash(new LinkedHashSet());
    TypeSets.checkTree(new TreeSet());
    TypeSets.check(new HashSet());           // HashSet<SetType> не работает нет hashCode()
    TypeSets.checkTree(new HashSet());      // HashSet<TreeType> не работает нет hashCode()
    TypeSets.check(new LinkedHashSet());
    TypeSets.checkTree(new LinkedHashSet());
    try {
        TypeSets.check(new TreeSet());      //не работает TreeSet<SetType> нет compareTo
    } catch (Exception e) {
    }
    try {
        TypeSets.checkHash(new TreeSet()); //не работает TreeSet<HashType> нет compareTo
    } catch (Exception e) {
    }
}

```

## SortedSet сортированные множества

- SortedSet сортированные множества [lesson\\_ch17/ch/ex09/include](#)
  - накладывают требования на наличие методов
- Методы
 

|                             |                                                       |
|-----------------------------|-------------------------------------------------------|
| ◦ Comparator comparator()   | возвращает Comparator<> для sorted, null для unsorted |
| ◦ Object first()            | наименьший элемент множества                          |
| ◦ Object last()             | наибольший элемент множества                          |
| ◦ SortedSet subSet(from,to) | подмножество                                          |
| ◦ SortedSet headSet(to)     | подмножество меньшее элемента < to                    |
| ◦ SortedSet tailSet(from)   | подмножество большее элемента > from                  |
- Пример. [lesson\\_ch17/ch/ex09/include](#)
- Пример. реализация sortedSet на базе LinkedList [lesson\\_ch17/ch/ex09/access](#)
- **ВНИМАНИЕ.** классный пример [lesson\\_ch17/ch/ex10/access](#)
- Пример. SortedList реализация Sorted дополнения LinkedList CheckSorted.app()
- Пример. SortedLinked реализация SortedSet интерфейса с Comparator CheckSorted.app2()
- 

## Queue LinkedList and PriorityQueue

- Очереди, есть две реализации, обычные на базе LinkedList и приоритетные на базе PriorityQueue
- Обычные очереди хранят элементы в порядке записи [lesson\\_ch17/ch/ex11/local](#)
  - LinkedList :one two three four five six seven eight nine ten
  - ArrayBlockingQueue :one two three four five six seven eight nine ten
  - ConcurrentLinkedQueue :one two three four five six seven eight nine ten
  - ConcurrentLinkedDeque :one two three four five six seven eight nine ten
- Приоритетные очереди хранят элементы в сортированном порядке [lesson\\_ch17/ch/ex11/local](#)
  - PriorityQueue :eight five four nine one seven six ten three two
  - PriorityBlockingQueue :eight five four nine one seven six ten three two

## Priority Queue

- PriorityQueue приоритетные очереди [lesson\\_ch17/ch/ex11/include](#)
- Пример. реализация приоритетной очереди
- ```
public class Item implements Comparable<Item>{ // реализует интерфейс Comparable<Item>
    private char primary;
    private int secondary;
    private String item;
    public Item( String item, char primary, int secondary) {
        this.item = item;
        this.primary = primary;
        this.secondary = secondary;
    }
    @Override
    public int compareTo(Item o) {
        if (primary < o.primary) {
            return -1;
        }
        if (primary == o.primary) {
            if (secondary < o.secondary) {
                return -1;
            } else if (secondary == o.secondary) {
                return 0;
            }
        }
        return 1;
    }
    @Override
    public String toString() {
        return Character.toString(primary) +
               secondary + ":" + item;
    }
}
public class PQList extends PriorityQueue<Item> {
    public void add(String item, char pri, int sec) { // использует класс <Item>
        super.add(new Item(item, pri, sec));
    }
}
```
- Применение
- ```
public static void app() {
    PQList pqList = new PQList();
    pqList.add("Empty trash", 'C', 4);
    pqList.add("Feed dog", 'A', 2);
    pqList.add("Feeedy bird", 'B', 7);
    pqList.add("Mow lawn", 'C', 3);
    pqList.add("Water lawn", 'A', 1);
    pqList.add("Feed Cat", 'B', 1);
    while (!pqList.isEmpty()) {
        System.out.println(pqList.remove());
    }
}
```

## Deque Деки

- Deque это двухсторонняя очередь
  - Deque очередь создается с помощью интерфейса <Deque> на базе LinkedList<>
- Пример. Очередь Deque на базе LinkedList который реализует интерфейс <Dequeue>

[lesson\\_ch17/ch/ex12/local](#)

```
public class CDeque<T> {  
    private LinkedList<T> deque = new LinkedList<T>();  
    public void addFirst(T e) {  
        deque.addFirst(e);  
    }  
    public void addLast(T e) {  
        deque.addLast(e);  
    }  
    public T getFirst() {  
        return deque.getFirst();  
    }  
    public T getLast() {  
        return deque.getLast();  
    }  
    public int size() {  
        return deque.size();  
    }  
    public T removeFirst() {  
        return deque.removeFirst();  
    }  
    public T removeLast() {  
        return deque.removeLast();  
    }  
    public static void fill(CDeque<Integer> deque) {  
        for (int i = 20; i < 27; i++) {  
            deque.addFirst(i);  
        }  
        for (int i = 50; i < 55; i++) {  
            deque.addLast(i);  
        }  
    }  
    @Override  
    public String toString() {  
        return deque.toString();  
    }  
}
```

- Применение

```
public static void app() {  
    CDeque<Integer> dq = new CDeque<>();  
    CDeque.fill(dq);  
    System.out.println(dq);  
    while (dq.size() > 0) {  
        System.out.print(dq.removeFirst() + " ");  
    }  
    System.out.println();  
    CDeque.fill(dq);  
    while (dq.size() > 0) {  
        System.out.print(dq.removeLast() + " ");  
    }  
    System.out.println();  
}
```

## Map Карты

- Map карты есть 6 реализаций интерфейса Map
  - HashMap, TreeMap, LinkedHashMap, WeakHashMap, ConcurrentHashMap, IdentityHashMap
- Методы
  - основные методы Map get() и put()
- Пример. простейший пример реализации Map
- ```
public class SimpleMap<K, V> { // самодельная карта Map
    private Object[][] pairs;
    private int index;
    public SimpleMap(int length) {
        pairs = new Object[length][2];
    }
    public void put(K key, V value) {
        if (index >= pairs.length) {
            throw new RuntimeException();
        }
        pairs[index++] = new Object[]{key, value};
    }
    public V get(K key) {
        for (int i = 0; i < index; i++) {
            if (key.equals(pairs[i][0])) {
                return (V) pairs[i][1];
            }
        }
        return null; // ничего не найдено
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < index; i++) {
            sb.append(pairs[i][0].toString());
            sb.append(":");
            sb.append(pairs[i][1].toString());
            if (i < index - 1) {
                sb.append("\n"); // везде кроме последней строки добавить \n
            }
        }
        return sb.toString();
    }
}
```
- Применение.
- ```
public static void app() {
    SimpleMap<String, String> map = new SimpleMap<>(6); // длину создаем сразу
    map.put("sky", "blue");
    map.put("grass", "green");
    map.put("ocean", "dancing");
    map.put("tree", "tall");
    map.put("earth", "brown");
    map.put("sun", "warm");
    try {
        map.put("extra", "object");
    } catch (Exception e) {
        System.out.println("catch: map bound exception");
    }
    System.out.println(map);
    System.out.println(map.get("ocean"));
}
```
-

## Производительность

- Производительность HashMap

## Типы Map и их свойства

| Map               | Описание                                                             |
|-------------------|----------------------------------------------------------------------|
| HashMap           | реализация Map на базе hash таблицы, поиск быстро и фиксирован       |
| LinkedHashMap     | тоже самое, работает немного медленнее, но хранит в порядке вставки  |
| TreeMap           | медленно, автоматом сортирует данные, только она имеет subMap()      |
| WeakHashMap       | HashMap позволяет GC удалять ключи, если на них нет внешних ссылок   |
| ConcurrentHashMap | HashMap многопоточных приложений, не использует синхро блокировку    |
| IdentityHashMap   | HashMap использует (==) вместо equals() применяется в особых случаях |

- HashMap использует хэширование как способ хранения данных
  - hashCode() для быстрого поиска по ключу
  - требования к ключам такие же как к <Set> для каждого ключа должен быть определен equals()
- ВНИМАНИЕ.** Properties работает как Map только через кастинг (Map)new Properties
- Пример. работа с Map и Properties

[lesson\\_ch17/ch/ex14/local](#)

## SortedMap

- SortedMap сортированная версия Map например TreeMap
- Методы Sorted Map
  - Comparator comparator()
  - Comparator comparator() возвращает Comparator<> для sorted, null для unsorted
  - T firstKey() наименьший элемент множества
  - T lastKey() наибольший элемент множества
  - SortedMap subMap(fromKey,toKey) подмножество Map
  - SortedMap headMap(toKey) подмножество Map меньшее Key < toKey
  - SortedMap tailMap(fromKey) подмножество Map большее Key > fromKey
- Пример. демонстрация методов SortedMap

[lesson\\_ch17/ch/ex15/local](#)

```
public static void app() {  
  
    TreeMap<Integer, String> sortedMap = new TreeMap<>(new CMapInt(10)); //сортированная  
    System.out.println(sortedMap);  
    Integer low = sortedMap.firstKey();  
    Integer high = sortedMap.lastKey();  
    System.out.println("low : " + low);  
    System.out.println("high: " + high);  
    Iterator<Integer> it = sortedMap.keySet().iterator();  
    for (int i = 0; i < 7; i++) {  
        if (i == 3) {  
            low = it.next();  
        } else if (i == 6) {  
            high = it.next();  
        } else {  
            it.next();  
        }  
    }  
    System.out.println("low : " + low);  
    System.out.println("high: " + high);  
    SortedMap sb = sortedMap.subMap(low, high);  
    SortedMap sb2 = sortedMap.headMap(high);  
    SortedMap sb3 = sortedMap.tailMap(low);  
}
```

## LinkedHashMap

- LinkedHashMap
  - работает почти также быстро как HashMap
  - хранит элементы в порядке добавления
  - можно настроить выборку в режим LRU (least recently used) редко используемые в начало списка
- Пример. включение режима LRU [lesson\\_ch17/ch/ex15/include](#)
- ```
public static void app() {
    LinkedHashMap<Integer, String> l hashmap = new LinkedHashMap<>(new CMapInt(9));
    System.out.println(l hashmap);
    System.out.println("LRU Order Demo:");
    l hashmap = new LinkedHashMap<Integer, String>(16, 0.75f, true);
    System.out.println(l hashmap);
    l hashmap.putAll(new CMapInt(9));
    System.out.println(l hashmap);
    for (int i = 0; i < 6; i++) {
        l hashmap.get(i); // имитация обращения
    }
    System.out.println(l hashmap); // перемещение элементов [0]..[6] в конец списка
    l hashmap.get(0);
    System.out.println(l hashmap); // свежий элемент [0] в самый конец
}
```

## HashCode и хэширование

- HashCode и хэширование
  - hash коды являются серьезным компонентом реализации HashMap
  - два метода одновременно работают с hash кодами это hashCode() и equals()
- **ВНИМАНИЕ.** Не забывать про реализацию hashCode() и переопределять вместе hashCode() и equals()
- При работе с HashMap
  - для работы с ключами определить hashCode() и equals()
- hashCode()
  - метод отвечает за распознавание ключей при работе с HashMap
  - метод определяется в классе хранения, базовая реализация hashCode() Object дает адрес ссылки
- **ВНИМАНИЕ.** Реализация hashCode() Object возвращает только адрес ссылки, и требует переопределения
- **ВНИМАНИЕ.** Методы hashCode() и equals() надо переопределять вместе в классе хранения
- equals()
  - правильный метод удовлетворяет следующим условиям
  - рефлексивность      для любого x                x.equals(y) возвращает true
  - симметричность      для любых x,y                x.equals(y) = true если y.equals(x) = true
  - транзитивность      для любых x,y,z            если x.equals(y)=true & y.equals(z)=true то x.equals(z)=true
  - стабильность          многократные                x.equals(y) возвращает всегда одно значение
  - отношение к null     для любого x != null        x.equals(null)   false
- Пример. [lesson\\_ch17/ch/ex15/integra](#)
  - Проблема: не распознается ключ в HashMap, хотя объект есть, поиском по ключу не находится
  - В классе хранения GroundHog не определены hashCode() и equals() и наследуются из Object
- Решение: надо переопределить hashCode() и equals() в классе хранения [lesson\\_ch17/ch/ex15/remote](#)
- **ВНИМАНИЕ.** instanceof проверяет <на принадлежность к классу> и <не является ли ссылка null>

## Понимание hashCode()

- Понимание hashCode()
- Качественное переопределение hashCode() и equals()
  - основная цель хэширования это обеспечить поиск по ключу
  - реализация hashCode() производится в классах полей объекта хранения и в классе Map.Entry
  - реализация equals() производится в классах полей объекта хранения и в классе Map.Entry
- Пример. полная реализация Map на базе двух ArrayList, класса AbstractMap<> и интерфейса Map.Entry<>
- **ВНИМАНИЕ.** toString() AbstractMap содержит как раз {}
- Реализация MapEntry<>

[lesson\\_ch17/ch/ex15/thrown](#)

```
public class MapEntry<K, V> implements Map.Entry<K, V> {  
    private K key;  
    private V value;  
    public SlowEntry(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
    @Override  
    public K getKey() {  
        return key;  
    }  
    @Override  
    public V getValue() {  
        return value;  
    }  
    @Override  
    public V setValue(V value) {  
        V oldValue = this.value;  
        this.value = value;  
        return oldValue;  
    }  
    @Override  
    public int hashCode() {  
        int keyHashCode = (key == null) ? 0 : key.hashCode(); //реализация ключа  
        int valueHashCode = (value == null) ? 0 : value.hashCode(); //реализация значения  
        return (keyHashCode ^ valueHashCode); // key XOR value  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof SlowEntry)) {  
            return false;  
        }  
        SlowEntry se = (SlowEntry) obj;  
        boolean keys, values;  
        if (key == null) {  
            keys = (se.getKey() == null);  
        } else {  
            keys = key.equals(se.getKey());  
        }  
        if (value == null) {  
            values = (se.getValue() == null); //  
        } else {  
            values = value.equals(se.getValue());  
        }  
        return (keys && values);  
    }  
    @Override  
    public String toString() { // toString обязательно в Entry  
        return key + "=" + value;  
    }  
}
```

- Реализация AbstractMap<K, V>

```
public class SlowMap<K, V> extends AbstractMap<K, V> {
    private List<K> keys = new ArrayList<>();
    private List<V> values = new ArrayList<>();
    @Override
    public V put(K key, V value) {
        V oldValue = get(key);
        if (!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else {
            values.set(keys.indexOf(key), value); // заменить значение
        }
        return oldValue;
    }
    @Override
    public V get(Object key) {
        if (!keys.contains(key)) {
            return null;
        }
        return values.get(keys.indexOf(key));
    }
    @Override
    public Set<Map.Entry<K, V>> entrySet() {
        Set<Map.Entry<K, V>> hSet = new HashSet<>();
        Iterator<K> ik = keys.iterator();
        Iterator<V> iv = values.iterator();
        while (ik.hasNext()) {
            hSet.add(new SlowEntry<K, V>(ik.next(), iv.next()));
        }
        return hSet;
    }
}
```

[lesson\\_ch17/ch/ex15/thrown](#)

- Применение

```
public static void app() {
    SlowMap<String, String> sMap = new SlowMap<>();
    sMap.putAll(Countries.capitals(15)); // проинициализировать
    System.out.println(sMap);
    System.out.println(sMap.get("EGYPT"));
    System.out.println(sMap.entrySet());
}
```

[lesson\\_ch17/ch/ex15/thrown](#)

## Реализация SlowMap на EntrySet и упрощенного MapEntry

- SlowMap на базе своего EntrySet и упрощенного MapEntry **??? РАЗОБРАТЬСЯ** [lesson\\_ch17/ch/ex16/local](#)
- **ВНИМАНИЕ.** Похоже все методы поиска и удаления работают через EntrySetIterator<K, V>
- Есть два способа добраться до функции keyset().removeAll()
  - реализовать свой keyset() что в принципе неверно, так как это заглушка
  - реализовать свой EntrySet это ВЕРНОЕ решение официальное

## Реализация EntrySet

- В SlowMap реализовано два варианта entryset() упрощенная и правильная
  - упрощенная реализована в SlowMap где просто заполняется Set<K, V> парами MapEntry
  - правильная реализация в SlowMap2 где создан полноценный EntrySet на базе AbstractSet<K, V>
- Реализация EntrySet
  - делается наследованием класса AbstractSet<Map.Entry<K, V>>
  - реализуются два метода iterator() и size() по умолчанию
  - внутри метода iterator() при создании класса new Iterator реализуется метод remove()
- **ВНИМАНИЕ.** именно метод newIterator().remove() отрабатывает keyset().remove(), keyset().removeAll()

- Реализация SlowMap на базе своего EntrySet

[lesson\\_ch17/ch/ex16/local](#)

```

public class SlowMap2<K, V> extends AbstractMap<K, V> {
    private List<K> keys = new ArrayList<>();
    private List<V> values = new ArrayList<>();
    private class EntrySet extends AbstractSet<Map.Entry<K, V>> {
        public Iterator<Entry<K, V>> iterator() {
            return new Iterator<Entry<K, V>>() {
                private int index = -1;
                public boolean hasNext() {
                    return index < keys.size() - 1;
                }
                public Entry<K, V> next() {
                    index++;
                    return new MapEntry<K, V>(keys.get(index), values.get(index));
                }
                public void remove() { // этот метод отрабатывает keyset().removeAll
                    if (index < 0) {
                        throw new RuntimeException("catch: can't remove");
                    }
                    keys.remove(index);
                    values.remove(index);
                    System.out.print("*");
                    index--;
                }
            };
        }
        public int size() {
            return keys.size();
        }
    }
    public Set<Entry<K, V>> entrySet() {
        return new EntrySet();
    }
    public V put(K key, V value) {
        V oldValue = get(key);
        if (!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else {
            values.set(keys.indexOf(key), value); // заменить значение
        }
        return oldValue;
    }
    public V get(Object key) {
        if (!keys.contains(key)) {
            return null;
        }
        return values.get(keys.indexOf(key));
    }
    public V remove(Object key) {
        if (!keys.contains(key)) {
            return null;
        }
        V oldValue = get(key);
        keys.remove(key);
        values.remove(oldValue);
        return oldValue;
    }
    public void clear() {
        keys.clear();
        values.clear();
    }
    public boolean isEmpty() {
        return (keys.isEmpty() && values.isEmpty());
    }
}

```

## Реализация SlowSet на базе AbstractSet

- SlowSet на базе AbstractSet lesson\_ch17/ch/ex18/access
- Есть два способа добраться до функции removeAll()
  - реализовать свою функцию removeAll() на базе AbstractSet() ВЫБРАНО ПО УМОЛЧАНИЮ
  - реализовать свою функцию remove() внутри Iterator<> которая отработает AbstractSet.removeAll()
- **ВНИМАНИЕ.** Что интересно функцию remove() можно задать внутри AbstractSet.iterator(), а add() нельзя lesson\_ch17/ch/ex18/access
- Пример. реализация Set
  - делается наследованием класса AbstractSet<K>
  - реализуются все методы, включая iterator() и size() по умолчанию
  - внутри метода iterator() при создании класса new Iterator реализуется метод remove()
  - либо в базовом наборе реализовать removeAll()
- ```
public class SlowSet<K> extends AbstractSet<K> {  
    private List<K> keys = new ArrayList<>();  
    @Override  
    public Iterator<K> iterator() {  
        return new Iterator<K>() {  
            private int index = -1;  
            @Override  
            public boolean hasNext() {  
                return index < keys.size() - 1;  
            }  
            @Override  
            public K next() {  
                index++;  
                return keys.get(index);  
            } // next  
        };  
    } // iterator  
    @Override  
    public boolean add(K k) {  
        return keys.add(k);  
    }  
    @Override  
    public boolean addAll(Collection<? extends K> c) {  
        return keys.addAll(c);  
    }  
    @Override  
    public boolean contains(Object o) {  
        return keys.contains(o);  
    } //contains  
    @Override  
    public boolean remove(Object o) {  
        return keys.remove(o);  
    } // remove  
    @Override  
    public boolean removeAll(Collection<?> c) {  
        return keys.removeAll(c);  
    }  
    @Override  
    public void clear() {  
        keys.clear();  
    } //clear  
    @Override  
    public int size() {  
        return keys.size();  
    } //size  
    @Override  
    public boolean isEmpty() {  
        return keys.isEmpty();  
    }  
}
```

## Hash for Speed

- Hash придуман как раз для того, чтобы ускорить процесс поиска в Map [lesson\\_ch17/ch/ex19/local](#)
- Примерный алгоритм поиска по hashCode()
  - весь массив значений разбивается на подмассивы меньшего размера
  - по ключу строится hashCode который указывает адрес такого подмассива
  - далее поиск в подмассиве идет «медленным способом» методом equals()
  - скорость достигается за счет быстрого получения hash code и размера подмассива 2..3 элемента
- Пример. простая реализация SimpleHashMap с полным набором команд [lesson\\_ch17/ch/ex23/access](#)
- 
- Пример. простая реализация SimpleHashSet с полным набором команд [lesson\\_ch17/ch/ex24/access](#)
- 
- Пример. Полная реализация SimpleHashMapL на своих EntrySet и LinkEntry [lesson\\_ch17/ch/ex25/access](#)
  - из за односвязного списка LinkEntry реализован свой полный EntrySet
  - все функции переписаны под односвязный список LinkEntry
- **ВНИМАНИЕ.** Из за односвязного списка пришлось реализовать честный EntrySet

## Переопределение hashCode() и

- Переопределение hashCode()
  - hashCode формируется в классе хранения и служит для индексирования узлов
  - индекс узла формируется уже в самой карте и зависит в том числе и от емкости таблицы узлов
- **ВНИМАНИЕ.** Class K в Map <K,V> ДОЛЖЕН иметь hashCode() и equals() чтобы работать в качестве ключа Map
- **ВНИМАНИЕ.** Обязательно переопределять вместе hashCode() и equals() чтобы они работали согласованно
- Пример индексирования узлов в SimpleHashMap [lesson\\_ch17/ch/ex25/access](#)
  - индекс узла создавался просто как остаток от деления на размер таблицы
  - чем больше размер таблицы и равномернее распределение hashCode, тем короче списки в узлах
- Требования к hashCode
  - hashCode должен строиться на важных свойствах, полях объекта, а не уникальном адресе ссылки
  - объекты с одинаковыми свойствами должны иметь равный hashCode независимо от размещения
  - hashCode должен распределяться равномерно, тогда не будет скучивания данных в узлах
- Алгоритм формирования hashCode по Джошуа Блош
  - сохранить любое ненулевое значение в переменной result = 17
  - значение каждого поля, вычисленное по таблице, прибавить к result\*37 и сохранить в result
  - повторять по циклу для всех значений полей, важных свойств объекта
  - проверить результат, что для одинаковых значений полей возвращается одинаковый hashCode()
- Метод подсчета hashCode      Примитивные типы      формулы получения значения для hashCode()
  - Поле      Формула      Примечание
  - Boolean       $c = (f ? 0 : 1)$
  - byte, char, short, int       $c = (\text{int})f$
  - long       $c = (\text{int})(f \wedge (f >> 32))$
  - float       $c = \text{Float.toFloatToIntBits}(f)$
  - double       $\text{long } l = \text{Double.doubleToLongBits}(f)$   
 $c = (\text{int})(f \wedge (f >> 32))$
  - Array      применить к каждому элементу
  - result = (result\*32 + c)       $c = \text{результат}, f = \text{исходное значение}$

| Поле                     | Формула                                                                                        | Примечание                 |
|--------------------------|------------------------------------------------------------------------------------------------|----------------------------|
| Boolean                  | $c = (f ? 0 : 1)$                                                                              |                            |
| byte, char, short, int   | $c = (\text{int})f$                                                                            | то есть целое значение     |
| long                     | $c = (\text{int})(f \wedge (f >> 32))$                                                         | xor верхней и нижней части |
| float                    | $c = \text{Float.toFloatToIntBits}(f)$                                                         | в целое число              |
| double                   | $\text{long } l = \text{Double.doubleToLongBits}(f)$<br>$c = (\text{int})(f \wedge (f >> 32))$ | тоже самое в целое число   |
| Array                    | применить к каждому элементу                                                                   |                            |
| result = (result*32 + c) | $c = \text{результат}, f = \text{исходное значение}$                                           |                            |

- Метод IDEA
  - $\text{result} = c;$
  - $\text{result} = \text{result} * 31 + c$

## Подсчет hashCode()

- Метод подсчета hashCode
- Пример. подсчет hashCode() на значениях String, int по формулам таблицы подсчета

[lesson\\_ch17/ch/ex26/local](#)

```
public class CountedString {  
    private static List<String> list = new ArrayList();  
    private String s;  
    private int id = 0;  
    public CountedString(String s) {  
        this.s = s;  
        list.add(s);  
        for (String s1 : list) { // на каждый экземпляр проверяет  
            if (s1.equals(s)) { // сколько раз одна и та же строка  
                id++; // попала в список  
            }  
        }  
    }  
    @Override  
    public int hashCode() {  
        int result = 17; // формируем hashCode() для объекта String  
        result = result * 37 + s.hashCode(); // поле s отработали  
        result = result * 37 + id; // поле id отработали  
        return result;  
    }  
    @Override  
    public boolean equals(Object o) { // версия сравнения с учетом String==null  
        if (this == o) return true;  
        if (!(o instanceof CountedString)) return false;  
        CountedString that = (CountedString) o;  
        if (id != that.id) return false;  
        return s != null ? s.equals(that.s) : that.s == null;  
    }  
    @Override  
    public String toString() {  
        return "[String:" + s + " id:" + id+ " hashCode():"+hashCode()+"]";  
    }  
}
```

- Реализация подсчета hashCode() класс Individual пакета Pets

[lesson\\_ch17/ch/ex26/include](#)

## Реализация hashCode() на базе IDEA

- Реализация hashCode() на базе IDEA мощная реализация классов Tuples
  - реализация hashCode() сделана по шаблону IDEA
- Реализация equals()
  - реализация equals() сделана по шаблону IDEA
  - реализация equals() в книге более прозрачная, но смысл один и тот же
  - **ВНИМАНИЕ.** проверка на класса нужна только если используется наследование
- Реализация compareTo()
  - реализация compareTo() сделана по книге
  - общий смысл вызывать compareTo() объектов хранения пока результат равен 0 и так до выхода
- **ВНИМАНИЕ.** мощная реализация классов Tuples

[lesson\\_ch17/ch/ex28/access](#)

## Container выбор реализации

- Существует 4 вида контейнеров Map, Set, List, Queue
  - для каждого типа реализации есть несколько видов реализации
- Выбор реализации для данного интерфейса определяется по требованиям к дизайну
  - самая быстрые реализации ArrayList, HashSet и HashMap их берут по умолчанию
  - если много вставок в середину списка выбирают LinkedList
  - для хранения в порядке вставки используют LinkedHashSet, LinkedHashMap
  - для сортировки по умолчанию TreeSet, TreeMap

## Среда Тестирования

- Среда тестирования контейнеров
  - класс предоставляет универсальный метод тестирования контейнеров разных классов
  - C = collection container T=TestParam
- Пример. class Test{}
  - в каждом объекте Test поле Test.name наименование теста
  - в каждом объекте Test абстрактный метод test(C,T) определяется при создании объекта Test{}
- Class TestParam
  - для каждого контейнера вызывается серия Test{} test(C,T) с параметрами TestParam{}
  - параметры это всегда список значений size размер container, loops число прогонов теста
- Class Tester
  - класс тестировщик, в него передаются объекты Test{} и параметры TestParam{}
  - программа Tester{} run() запускает серию тестов

## Выбор и тестирование с List

- Работа и тестирование с List
    - переопределяет функцию initialize() очистка коллекции перед тестами
    - запускает run из созданного экземпляра наследника, таким образом цепляет новую initialize()
  - ListTester тестирование реализаций List


|              |            |                      |                                    |
|--------------|------------|----------------------|------------------------------------|
| ◦ ArrayList  | ListTester | initialize() новая   | container очищается в initialize() |
| ◦ LinkedList | ListTester | initialize() новая   | container очищается в initialize() |
| ◦ Queue      | Tester     | initialize() базовая | container очищается в каждом teste |
- Результаты тестирования
  - ArrayList медленно вставляет и удаляет в середине списка
  - LinkedList медленно выполняет set и get
  - Queue быстро выполняет свои операции
- Пример. реализация List<Integer>
  - lesson\_ch17/ch/ex29/local
- Пример. реализация List<String>
  - lesson\_ch17/ch/ex29/access

## Выбор и тестирование с List

- Работа и тестирование с List
  - lesson\_ch17/ch/ex29/local
- Пример. подключение методов ArrayList и LinkedList одновременно
  - lesson\_ch17/ch/ex33/access
- **ВНИМАНИЕ.** Общий смысл вставить семафор и пересоздавать ArrayList|LinkedList при переключении
  - решение предполагает, что семафор переключается редко и затем идет серия однотипных команд
  - в официальном решении создано два объекта ArrayList и LinkedList равноправных
  - у меня список наследует LinkedList и для методов get/set пересоздает ArrayList по семафору

## Опасности микротестов

- Опасности микротестов
  - Random() включают граничные значения 0.0 и 1.0 надо быть осторожным и учесть это в тестах
  -

## Выбор и тестирование с Set

- Работа и тестирование с Set [lesson\\_ch17/ch/ex34/include](#)
  - HashSet           работает быстрее всех
  - TreeSet          работает медленнее в целом, <Iterate> быстрее HashSet, медленнее LinkedHashSet
  - LinkedHashSet   немного медленее чем HashSet, однозначно быстрее TreeSet

## Выбор и тестирование с Map

- Работа и тестирование с Map [lesson\\_ch17/ch/ex35/local](#)
  - все Map замедляются при увеличении объема Map
  - HashMap         самая быстрая Map, использовать по умолчанию
  - можно создать на базе TreeMap методом putAll(TreeMap)
  - TreeMap         самая медленная Map, из за встроенной сортировки
  - очень удобна для экспорта ключей/значений keyset(), values() в массивы
  - Arrays.binarySearch()   быстро отработает поиск в таких массивах
  - можно создать на базе HashMap методом putAll(HashMap)
  - LinkedHashMap    медленнее HashMap, быстрее остальных, хранит данные в порядке вставки
  - IdentityHashMap    медленная за счет сравнения объектов (==)
  - WeakHashMap
- Пример. реализация SlowMapEntry на базе List<Entry<K,V>> с компаратором [lesson\\_ch17/ch/ex36/access](#)
- Пример. реализация SlowMapEntrySort на базе List<Entry<K,V>> с сортировкой, резко ускорен метод get()
- **ВНИМАНИЕ.** Мощные примеры используют неполное сравнение и компаратор в объекте хранения
- чтобы реализовать Map на базе ArrayList<Entry<K,V>>
- **ВНИМАНИЕ.** Class K в Map <K,V> ДОЛЖЕН иметь hashCode() и equals() чтобы работать в качестве ключа Map

## Факторы влияющие на производительность HashMap

- Факторы влияющие на производительность HashMap
  - Capacity       емкость, количество узлов Hash Table
  - Initial capacity начальная емкость HashTable
  - Size           число заполненных узлов HashTable
  - Load Factor   коэффициент загрузки это отношение size/capacity оптимальное значение 0.75
  - незагруженная таблица мало коллизий, загруженная коллизии и поиск в узлах
- Перехэширование
  - если загрузка HashMap превысила 0.75, то создают новую HashMap большего размера
  - затем копируют в тело новой HashMap старую, это называется «перехэширование»
- Пример. Получение Load Factor и тестирование HashMap с разной загрузкой [lesson\\_ch17/ch/ex38/access](#)
- **ВНИМАНИЕ.** Мощный пример получения Load Factor через Reflection
- Пример. Реализация функции rehash() [lesson\\_ch17/ch/ex39/access](#)
- **ВНИМАНИЕ.** Мощный пример реализации функции rehash() и алгоритма ПРОСТЫХ ЧИСЕЛ
-

## Вспомогательные методы Collection

- Вспомогательные методы Collection

- 

| Метод                             | Описание                                       | Примечание |
|-----------------------------------|------------------------------------------------|------------|
| checkedCollection()               | представление Collection которое               |            |
| checkedList()                     | обеспечивает динамическую                      |            |
| checkedMap()                      | безопасность типов                             |            |
| checkedSet()                      |                                                |            |
| checkedSortedMap()                |                                                |            |
| checkedSortedSet()                |                                                |            |
| max() , min()                     | наибольший и наименьший элемент                |            |
| max(Collection, Comp)             | тоже самое с компаратором                      |            |
| min(Collection, Comp)             |                                                |            |
| indexOfSublist(List, SubList)     | первое вхождение SubList                       |            |
| lastIndexOfSublist(List, SubList) | последнее вхождение SubList                    |            |
| replaceAll()                      | замена всех вхождений                          |            |
| reverse(List)                     | реверс коллекции                               |            |
| reverseOrder()                    | возвращает компаратор реверса                  |            |
| reverseOrder(Comp)                | заменяет компаратор реверса                    |            |
| rotate()                          | ротация коллекции                              |            |
| shuffle()                         | перемешивает коллекцию                         |            |
| shuffle(List, Random)             | тоже но со своим Random()                      |            |
| sort()                            | сортировка                                     |            |
| sort(List, Comp)                  | сортировка с внешним компаратором              |            |
| copy()                            | копирует коллекцию                             |            |
| swap()                            | меняет местами элементы коллекции              |            |
| fill()                            | заполняет коллекцию элементом                  |            |
| nCopies()                         | возвращает коллекцию из n копий элемента       |            |
| disjoint()                        | true если коллекции разные                     |            |
| frequency()                       | число повторов элемента в коллекции            |            |
| emptyList()                       | true если коллекция пустая                     |            |
| emptyMap()                        |                                                |            |
| emptySet()                        |                                                |            |
| singleton()                       | вернуть неизменный экземпляр Set               |            |
| singletonList()                   | вернуть неизменный экземпляр List              |            |
| singletonMap()                    | вернуть неизменный экземпляр Map               |            |
| list(Enumeration)                 | вернуть ArrayList с порядком как в Enumeration |            |
| enumeration()                     | создает Enumeration в старом стиле             |            |

- 

- Пример. реализация всех указанных методов

[lesson\\_ch17/ch/ex40/local](#)

-

## Сортировка и поиск в списках

- Сортировка и поиск в списках
  - сортировку и поиск в коллекциях делают статические методы Collections
- **ВНИМАНИЕ.** Если Comparator использовался при вызове sort(), его же надо использовать в binarySearch()
- Пример. реализация сортировки с компаратором List
- Пример. сортировка с компаратором Set для Map
- Пример. сортировка с компаратором List по алфавиту (toLowerCase)
- **ВНИМАНИЕ.** Мощный пример.
- Сортировка и поиск в списках

[lesson\\_ch17/ch/ex40/include](#)

[lesson\\_ch17/ch/ex40/access](#)

[lesson\\_ch17/ch/ex41/access](#)

[lesson\\_ch17/ch/ex40/access](#)

## Получение неизменяемых Collections and Map

- Получение неизменяемых Collections and Map
- Методы Collections.unmodifiable()
  - это способ создания коллекций только для чтения
  - применяется для закрытых коллекций, внутри класса их можно изменить
  - но пользователям наружу выдается неизменная «read only» копия
- Пример. реализация unmodifiable Collection

[lesson\\_ch17/ch/ex42/local](#)

[lesson\\_ch17/ch/ex42/local](#)

## Синхронизация Collection or Map

- Синхронизация Collection or Map
- Методы Collections.synchronized()
  - синхронизация, принудительный метод для работы с Collection или Map между потоками
  - синхронизацию надо выполнять сразу же как созданы Collection или Map
- 

[lesson\\_ch17/ch/ex42/include](#)

## Fail Fast Срочный отказ

- Срочный отказ
  - это отказ в обслуживании когда во время работы итератора меняется коллекция его вызывавшая
- Решение. Обновить итератор после изменения Collection и не менять Collection пока работает Iterator<>

[lesson\\_ch17/ch/ex42/integra](#)

## Удержание ссылок

- Удержание ссылок
  - используется для освобождения памяти объектами с низким приоритетом
- Три класса удержания ссылок SoftReference, WeakReference, PhantomReference
  - по сути это контроль степени ослабления ссылок перед GC
  - SoftReference самый сильный, PhantomReference самый слабый класс ссылок
- Теория
  - если объект имеет ссылку то GC не может его удалить
  - если между объектом и GC подключен класс удержания ссылки, то GC МОЖЕТ удалить объект
  - используется ReferenceQueue механизм освобождения ресурсов объекта перед удалением
- SoftReference используется для буферов памяти
  - можно выбрать помещать объект в ReferenceQueue
- WeakReference используется для реализации «канонического отображения» ???
  - можно выбрать помещать объект в ReferenceQueue
- PhantomReference используется для контролируемого удаления finalize() объекта
  - объект автоматом помещается в ReferenceQueue
- **ВНИМАНИЕ.** Классная тема, работа с объектами которые могут исчезнуть в любой момент

[lesson\\_ch17/ch/ex42/remote](#)

## WeakHashMap

- WeakHashMap
    - специальная карта для хранения слабых ссылок, которые может удалить GC
    - создана для реализации «канонического отображения» ???
  - Теория
    - ключи и данные помещаются в WeakHashMap и автоматом оборачиваются в WeakReference
    - для освобождения памяти вызывается GC, а метод WeakHashMap.Item.finalize() проводит очистку
  - **ВНИМАНИЕ.** Class K в Map <K,V> ДОЛЖЕН иметь hashCode() и equals() чтобы работать в качестве ключа Map
  - Пример. реализация WeakHashMap
  - ```
public class Element {  
    private String ident;  
    public Element(String ident) {  
        this.ident = ident;  
    }  
    @Override  
    public boolean equals(Object o) { // реализация для работы в качестве ключа Map  
        if (this == o) return true;  
        if (!(o instanceof Element)) return false;  
        Element element = (Element) o;  
        return ident != null ? ident.equals(element.ident) : element.ident == null;  
    }  
    @Override  
    public int hashCode() { // реализация для работы в качестве ключа Map  
        return ident != null ? ident.hashCode() : 0;  
    }  
    @Override  
    protected void finalize() {  
        System.out.println("Finalizing:" + ident);  
    }  
    @Override  
    public String toString() {  
        return ident;  
    }  
}  
• public class Key extends Element { // класс реализация ключа Key Map<KeyValue>  
    public Key(String ident) {  
        super(ident);  
    }  
}  
• public class Value extends Element { // класс реализация значение Value Map<KeyValue>  
    public Value(String ident) {  
        super(ident);  
    }  
}
```
  - Применение. создается карта WeakHashMap, после создания вызывается GC
    - каждая 2я ссылка прописывается дополнительно в массиве key[] как обычные ссылки
    - в результате GC затирает только нечетные элементы
- ```
public static void check() {  
    int size = 10;  
    Key[] keys = new Key[size];  
    WeakHashMap<Key, Value> wMap = new WeakHashMap<>();  
    for (int i = 0; i < size; i++) {  
        Key k = new Key("Key" + i);  
        Value v = new Value("Value" + i);  
        if (i % 2 == 0) {  
            keys[i] = k; // сохраняется каждая 2 ссылка как обычная GC их не трогает  
        }  
        wMap.put(k, v);  
    }  
    System.gc(); // вызывается GC для освобождения wMap  
}
```

[lesson\\_ch17/ch/ex42/thrown](#)

[lesson\\_ch17/ch/ex42/thrown](#)

## Collections Java 1.0, Java 1.1

- Collections Java 1.0, Java 1.1
  - это устаревшие Collection которые могут встретиться в библиотеках

## Vector и Enumeration

- Vector и Enumeration
  - это два типа коллекций, которые уже не используются
  - Vector заменен ArrayList<>
  - Enumeration заменен Iterator{}
- Конвертирования Vector и Enumeration в современные коллекции и обратно
  - для их конвертирования в современные применяются методы Collection.enumeration()
- Пример. реализация конвертации данных между Vector и ArrayList [lesson\\_ch17/ch/ex42/unit](#)

## HashTable и Stack

- HashTable и Stack
  - HashTable заменен HashMap
  - Stack заменен LinkedList
  - Stack является производным от Vector, все команды Vector работают с Stack
- Пример. реализация стека на Stack, LinkedList [lesson\\_ch17/ch/ex42/unit](#)
  - LinkedList toString() LIFO, выводит данные LIFO
  - Stack toString() FIFO, выводит данные LIFO

## BitSet

- BitSet замещен EnumSet [lesson\\_ch17/ch/ex42/value](#)
  - предпочтительно использовать EnumSet
  - BitSet рекомендуется использовать если длина данных неизвестна и нужны операции BitSet
- BitSet
  - класс для хранения битовых данных,
  - минимальная длина данных 64 бита на случай если надо сохранить всего 8 бит флагов
  - максимальная длина неограничена ничем.
- **ВНИМАНИЕ.** Не надо экономить память и тратить время на свой класс хранения битов если данных мало
- Решение о применении BitSet или своего класса принимать только по профилю и затратам
- Пример. реализация использования класса BitSet [lesson\\_ch17/ch/ex42/value](#)
  - длина наращивается адаптивно, в зависимости от установленных бит данных
  - при сбросе битовых данных длина остается той же что и была.

# I/O System

## Class File

- ClassFile
    - работает с файлами или каталогами
    - метод File.list() выдает список файлов каталога
    - интерфейс FilenameFormatter используется совместно с классом File
  - Работа с каталогами
  - FilenameFilter интерфейс фильтра для чтения групп файлов
    - pattern regex.pattern для сортировки полученных файлов
    - accept() метод который сортирует файлы по фильтру regex
  - Пример. реализация фильтра \*.zip в трех вариантах
- [lesson\\_ch18/ch/ex01/local](#)
- Пример. реализация на внешнем классе
  - ```
public class DirFilter implements FilenameFilter{ // фильтр для имён файлов
    private Pattern pattern;
    private String regex;
    public DirFilter(String regex) {
        this.regex = regex;
        this.pattern = Pattern.compile(regex);
    }
    @Override
    public boolean accept(File dir, String name) { // классически файлы по pattern
        return pattern.matcher(name).matches();
    }
}
public class DirList {
    public static void getDir(String[] args) {
        File path = new File(".");
        String[] list; // массив для хранения
        if (args.length == 0) {
            list = path.list(); // получить список каталогов
        } else {
            list = path.list(new DirFilter(args[0])); // искать каталоги по pattern
        }
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER); // сортировать каталог
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```
  - Применение
  - ```
public static void app(String[] args) {
    // String regex = "(\w+\W+)+zip"; // выбранные
    String regex = ".+"; // все файлы
    System.out.println("Dir with External Class:");
    System.out.println("-----");
    DirList.getDir(new String[]{regex}); // вместо args[]
}
```

- Пример. реализация на внутреннем классе

[lesson\\_ch18/ch/ex01/local](#)

```
public class DirList2 {
    public static FilenameFilter filter (final String regex) { // реализация интерфейса
        return new FilenameFilter() {
            private Pattern pattern = Pattern.compile(regex); // скомпилировали regex
            @Override
            public boolean accept(File dir, String name) {
                return pattern.matcher(name).matches();
            }
        };
    }
    public static void getDir(String[] args) {
        File path = new File("."); // путь до каталога
        String[] list; // массив для хранения
        if (args.length == 0) {
            list = path.list(); // получить список каталогов
        } else {
            list = path.list(filter(args[0])); // искать каталоги по pattern
        }
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER); // сортировать каталог
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

- Применение

```
public static void app(String[] args) {
//    String regex = "(\w+\W+)+zip"; // выбранные
    String regex = ".+"; // все файлы
    System.out.println("\nDir with Internal Class:");
    DirList2.getDir(new String[]{regex});
}
```

- Пример. реализация на методе с внутренним классом

[lesson\\_ch18/ch/ex01/local](#)

- **ВНИМАНИЕ.** args[0] из внешнего класса используемый в внутреннем классе метода должен быть FINAL
- **ВНИМАНИЕ.** regex тут работает ТОЛЬКО для локального каталога файлов, для рекурсии ДРУГАЯ стратегия

```
public class DirList3 {
    public static void getDir(String[] args) {
        File path = new File("."); // путь до каталога
        String[] list; // массив для хранения
        if (args.length == 0) {
            list = path.list(); // получить список каталогов
        } else {
            list = path.list(new FilenameFilter() { // работает только с final arg
                private Pattern pattern = Pattern.compile(args[0]);
                @Override
                public boolean accept(File dir, String name) {
                    return pattern.matcher(name).matches();
                } // искать каталоги по pattern
            });
        }
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER); // сортировать каталог
        for (String s : list) {
            System.out.println(s);
        }
    }
}

public static void app(String[] args) {
//    String regex = "(\w+\W+)+zip"; // выбранные
    String regex = ".+"; // все файлы
    System.out.println("\nDir with Method Internal Class:");
    DirList3.getDir(new String[]{regex});
}
```

## Средства работы с каталогами

- Средства работы с каталогами
  - local() метод работает с локальными файлами в одном каталоге
  - walk() метод проходит все дерево каталогов
- Пример. реализация рекурсивного чтения каталогов

```
public class Catalog {  
    public static class TreeInfo implements Iterable<File> { // класс работы с деревом  
        private List<File> listFiles = new ArrayList<>();  
        private List<File> listDirs = new ArrayList<>();  
        @Override  
        public Iterator<File> iterator() { // по умолчанию Iterator() listFiles<>  
            return listFiles.iterator();  
        }  
        private void addAll(TreeInfo treeInfo) { // поглотить коллекцию  
            listFiles.addAll(treeInfo.listFiles);  
            listDirs.addAll(treeInfo.listDirs);  
        }  
        @Override  
        public String toString() {  
            return "Catalogs:" + PPrint.pFormat(listDirs) + "\n"+  
                "Files : " + PPrint.pFormat(listFiles);  
        }  
    }  
    public static File[] local(File path, final String regex) { // regex must be final  
        return path.listFiles(new FilenameFilter() { // to use it in the inner class  
            private Pattern pattern = Pattern.compile(regex); // final helps to do this  
            @Override  
            public boolean accept(File dir, String name) {  
                return pattern.matcher(new File(name).getName()).matches(); // работает  
//                    return pattern.matcher(name).matches(); // работает точно также ???  
            }  
        });  
    } // local()  
    public static File[] local(String path, final String regex) { // перегрузка  
        return local(new File(path), regex);  
    }  
    private static TreeInfo recursive(File path, String regex) {  
        TreeInfo treeInfo = new TreeInfo(); // конструктор по умолчанию  
        for (File file : path.listFiles()) { // прогнать базовый каталог  
            if (file.isDirectory()) {  
                treeInfo.listDirs.add(file); // добавить каталог в папку каталогов  
                TreeInfo subTree = recursive(file, regex); //папки и файлы по regex  
                treeInfo.addAll(subTree); // поглотить внутренние каталоги и файлы  
            } else {  
                if (file.getName().matches(regex)) {  
                    treeInfo.listFiles.add(file);  
                }  
            }  
        }  
        return treeInfo;  
    }  
    public static TreeInfo walk(File path, String regex) {  
        return recursive(path, regex);  
    }  
    public static TreeInfo walk(String path, String regex) { // перегружен (path, ".")  
        return recursive(new File(path), regex);  
    }  
}
```
- Применение

## Strategy Pattern для рекурсивной обработки файлов

- Strategy Pattern для рекурсивной обработки файлов
  - интерфейс Strategy используется для альтернативной обработки дерева каталогов
  - возможно создавать новую версию прямо в вызове метода и внутреннего класса
- Пример. реализация паттерна Strategy [lesson\\_ch18/ch/ex04/include](#)

```
public interface IStrategy {  
    void process(File file);  
}  
public class ProcessFiles {  
    private IStrategy strategy; // обработчик файлов IStrategy.process(File)  
    private String ext;  
  
    public ProcessFiles(IStrategy strategy, String ext) {  
        this.strategy = strategy;  
        this.ext = ext;  
    }  
    public void processTree(File root) throws IOException{ // walk обеспечивает рекурсию  
        for (File file : Catalog.walk(root.getAbsolutePath(),".*\\\\"+ext)) {  
            strategy.process(file.getCanonicalFile());  
        } // с полным путем все файлы Tree данного расширения  
    } // file.getCanonicalFile() убирает из пути ".." маркер текущего каталога  
    public void start(String[] args) {  
        try {  
            for (String fileName : args) {  
                File file = new File(fileName);  
                if (file.isDirectory()) {  
                    processTree(file); // issue IOException  
                } else {  
                    if (!fileName.endsWith("." + ext)) {  
                        fileName += "."+ext;  
                    strategy.process(new File(fileName).getCanonicalFile());  
                }  
            }  
        } catch (IOException e) {  
            throw new RuntimeException(e); // выдать ошибку ввода вывода  
        }  
    }  
}
```

- Применение. реализация обработчика дерева каталогов

```
public static void app(String[] args) {  
    System.out.println("Long version:");  
    IStrategy stg = new IStrategy() { // объект интерфейса  
        @Override  
        public void process(File file) {  
            System.out.println(file);  
        }  
    };  
    ProcessFiles pf = new ProcessFiles(stg, "java");  
    pf.start(args);  
  
    System.out.println("\nShort version:");  
    new ProcessFiles(new IStrategy() { // анонимный класс обработчика файлов  
        @Override  
        public void process(File file) {  
            System.out.println(file);  
        }  
    }, "java").start(new String[]{"./src/ch18"}); // внутри же метода массив каталогов  
}
```

## Проверка и создание каталогов

- Проверка и создание каталогов
- Пример. реализация создания каталогов
- Пример. реализация проверки файлов по времени создания

[lesson\\_ch18/ch/ex06/local](#)  
[lesson\\_ch18/ch/ex06/access](#)

## Ввод и вывод

- Потоки stream данных это классы ОДНОБАЙТОВЫЙ ввод вывод данных
  - stream это абстракция, которая скрывает низкоуровневый доступ к внешним устройствам
  - поток бывают вввода данных InputStream и вывода данных OutputStream
  - программист редко работает с потоками напрямую через методы read(), write()

## InputStream

- Источники InputStream
  - массив байт
  - строка
  - файл
  - pipe канал работает как трубопровод, данные помещаются в один конец, считаются с другого
  - последовательность различных потоков
  - источники данных из Интернет или сети
- Классы InputStream

| Класс                   | Описание                                           | Примечание |
|-------------------------|----------------------------------------------------|------------|
| ByteArrayInputStream    | Буфер данных как источник данных                   |            |
| StringBufferInputStream | Строка как источник данных                         |            |
| FileInputStream         | Файл как источник данных                           |            |
| PipedInputStream        | PipedOutputStream как источник данных              |            |
| SequenceInputStream     | Сливает два и более потоков в один источник данных |            |
| FilterInputStream       | Абстрактный класс интерфейс для потоков            |            |

## OutputStream

- Приемники OutputStream
  - массив байт
  - файл
  - pipe канал работает как трубопровод, данные помещаются в один конец, считаются с другого
  - последовательность различных потоков
  - источники данных из Интернет или сети
- Классы OutputStream

| Класс                 | Описание                                | Примечание |
|-----------------------|-----------------------------------------|------------|
| ByteArrayOutputStream | Буфер данных как приемник данных        |            |
| FileOutputStream      | Файл как приемник данных                |            |
| PipedOutputStream     | PipedInputStream как приемник данных    |            |
| FilterOutputStream    | Абстрактный класс интерфейс для потоков |            |

## Добавление атрибутов и интерфейсов Декораторы Потоков

- Добавление атрибутов и интерфейсов с помощью классов декораторов
  - декоратор FilterInputStream добавляет функционал в InputStream классы
  - декоратор FilterOutputStream добавляет функционал в OutputStream классы
- Декораторы FilterInputStream

| Класс                 | Описание                                         | Примечание |
|-----------------------|--------------------------------------------------|------------|
| DataInputStream       | вмест с DataOutputStream работает с примитивами  |            |
| BufferedInputStream   | используется для буферизации чтения с устройства |            |
| LineNumberInputStream | следит за числом считанных с устройства строк    |            |
| PushBackInputStream   | использует односимвольный буфер                  |            |

- Декораторы FilterOutputStream

| Класс                | Описание                                       | Примечание |
|----------------------|------------------------------------------------|------------|
| DataOutputStream     | вмест с DataInputStream работает с примитивами |            |
| PrintStream          | форматирование данных на вывод                 |            |
| BufferedOutputStream | используется для буферизация вывода данных     |            |

## Классы Reader и Writer

- Классы Reader и Writer это классы которые работают с СИМВОЛАМИ UNICODE данных
- Источники и приемники данных соответствие потокам InputStream и OutputStream

| Класс Stream            | Класс Reader, Writer                | Примечание |
|-------------------------|-------------------------------------|------------|
| InputStream             | Reader адаптер : InputStreamReader  |            |
| OutputStream            | Writer адаптер : OutputStreamWriter |            |
| FileInputStream         | FileReader                          |            |
| FileOutputStream        | FileWriter                          |            |
| StringBufferInputStream | StringReader                        |            |
| ---                     | StringWriter                        |            |
| ByteArrayInputStream    | CharArrayReader                     |            |
| ByteArrayOutputStream   | CharArrayWriter                     |            |
| PipedInputStream        | PipedReader                         |            |
| PipedOutputStream       | PipedWriter                         |            |

## Изменение поведения потока декораторы для Reader , Writer классов

- Декораторы Reader , Writer классов

| Класс Stream         | Класс Декораторы Reader, Writer           | Примечание |
|----------------------|-------------------------------------------|------------|
| FilterInputStream    | FilterReader                              |            |
| FilterOutputStream   | FilterWriter                              |            |
| BufferedInputStream  | BufferedReader                            |            |
| BufferedOutputStream | BufferedWriter                            |            |
| DataInputStream      | DataInputStream, для строк BufferedReader |            |
| PrintStream          | PrintWriter                               |            |
| StreamTokenizer      | StreamTokenizer                           |            |
| PushBackInputStream  | PushBackReader                            |            |

- 
- **ВНИМАНИЕ.** Для строк ВМЕСТО DataInputStream, ИСПОЛЬЗОВАТЬ BufferedReader
- **ВНИМАНИЕ.** Во всех остальных случаях использовать DataInputStream
- **ВНИМАНИЕ.** Вместо PrintStream ИСПОЛЬЗОВАТЬ PrintWriter, они идентичны, PrintWriter более продвинут

## Класс RandomAccessFile

- Класс RandomAccessFile не является частью структуры InputStream и OutputStream
  - это полностью независимый класс, реализует интерфейсы DataInput, DataOutput
  - только в этом классе возможно прямое позиционирование

## Типовое использование потоков ввода вывода

- Буферизованное чтение из файла
  - реализуется на базе класса FileReader
- Пример. реализация буферизованного чтения по строкам readLine()
- **ВНИМАНИЕ.** readLine() отбрасывает "\n" поэтому нужен StringBuilder

[lesson\\_ch18/ch/ex06/local](#)

## Буферизованный ввод применение

- Задача. Чтение параметров классов и создание экземпляров внутренних классов с параметрами
- Проблема. Класс нельзя инициализировать до непосредственного использования
- Решение. Двухэтапная инициализация
  - в Map хранятся объекты Factory<>, которые могут вызвать метод create() с параметрами
  - при наличии совпадения, Factory<>.create( params) и создается живой экземпляр
  - **ВНИМАНИЕ.** метод Factory<>.crcreate(param) запоминает параметры в полях Factory<> при повторном вызове Factory<>.create() используются запомненные параметры

- Пример. Чтение параметров и реализация двухэтапного паттерна Factory [lesson\\_ch18/ch/ex11/access](#)
- Интерфейс Factory<>

```
public interface IFactoryGC< C extends GreenHouse, T extends Event> {  
    T create();  
    T create(C c, long delay); // сразу создаем объект внутри класса  
    T create(C c, long delay, int count, List<T> list); // на два параметра  
}
```

- Классы Factory<> чтение параметров [lesson\\_ch18/ch/ex11/access](#)

```
public class GCFfileRead {  
    private static Map<String, IFactoryGC<GreenHouse, Event>> hMap = new HashMap<>();  
    static {  
        hMap.put("LightOn".toLowerCase(), new GCFfactory.LightOn());  
        hMap.put("Bell".toLowerCase(), new GCFfactory.Bell());  
        hMap.put("Restart".toLowerCase(), new GCFfactory.Restart());  
    }  
    public static void events(GreenHouse gc, String fileName) {  
        List<Event> list = new ArrayList<>();  
        List<String[]> glist = new ArrayList<>();  
        try {  
            BufferedReader in = new BufferedReader(new FileReader(fileName));  
            String s;  
            while ((s = in.readLine()) != null) { // читаем строку из файла в s  
                String[] ss = s.replaceAll("[/ ]", "").split("\\\\W+");  
                if (ss == null || ss.length < 4 || (!ss[0].equals("event") &&  
                    !ss[0].equals("system"))) {  
                    continue;  
                }  
                IFactoryGC<GreenHouse, Event> factEvent = hMap.get(ss[1].toLowerCase());  
                if (factEvent == null) {  
                    continue;  
                }  
                if (ss[0].equals("system")) {  
                    glist.add(ss); // отрабатываем только созранение массива list  
                } else {  
                    list.add(factEvent.create(gc, Long.valueOf(ss[2])));  
                }  
            }  
            in.close(); // закрываем поток  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
        for (String[] ss : glist) { // системные вызовы  
            gc.addEvent(hMap.get(ss[1].toLowerCase()).create(gc,  
                Long.valueOf(ss[2]), Integer.valueOf(ss[3]), list));  
        }  
    }  
}
```

- Применение [lesson\\_ch18/ch/ex11/access](#)

```
public static void app() {  
    HumidityControl gc = new HumidityControl(); // создали объект оранжерея  
    GCFfileRead.events(gc, "./src/ch18/ex11/access/EventsList.java"); // прочитали  
    gc.run(); // запустили  
}
```

- Классы Factory<>

lesson\_ch18/ch/ex11/access

- **ВНИМАНИЕ.** Создание объектов в два этапа сначала создается с параметрами, потом create() копия

```

abstract public class GCFactory implements IFactoryGC<GreenHouse, Event> {
    protected GreenHouse gc;
    protected long delay;
    protected int count;
    abstract public Event create();
    public Event create(GreenHouse greenHouse, long delay) {
        return null;
    }
    public Event create(GreenHouse greenHouse, long delay, int count, List<Event> list) {
        return null;
    }
    protected void initParam(GreenHouse gc, long delay) {
        if (gc == null || delay == 0) {
            throw new RuntimeException("Call not initialized object");
        }
        this.gc = gc;
        this.delay = delay;
        this.count = 1; // заглушка
    }
    protected void checkParam() {
        if (gc == null || delay == 0 || count == 0) {
            throw new RuntimeException("Call not initialized object");
        }
    }
    public static class LightOn extends GCFactory {
        public Event create() {
            checkParam();
            return gc.new LightOn(delay);
        }
        public Event create(GreenHouse gc, long delay) {
            initParam(gc, delay);
            return gc.new LightOn(delay);
        }
    }
    public static class Bell extends GCFactory {
        public Event create() {
            checkParam();
            return gc.new Bell(delay);
        }
        public Event create(GreenHouse gc, long delay, int count, List<Event> list) {
            initParam(gc, delay);
            return gc.new Bell(delay, count);
        }
    }
    public static class Restart extends GCFactory {
        private Event[] events;
        public Event create() {
            checkParam();
            if (events == null) {
                throw new RuntimeException("Call not properly initialized Object");
            }
            return gc.new Restart(delay, events);
        }
        public Event create(GreenHouse gc, long delay, int count, List<Event> list) {
            initParam(gc, delay);
            if (list == null) {
                throw new RuntimeException("Call not properly initialized Object");
            }
            events = list.toArray(new Event[0]);
            return gc.new Restart(delay, events);
        }
    }
}

```

## Чтение из памяти

- Чтение из памяти
  - StringReader чтение строки в символы
  - read() читает один символ за раз в переменную int
  - read(char[]) читает в буфер char[] группу символов
- Пример. реализация посимвольного чтения строки [lesson\\_ch18/ch/ex12/classa](#)

```
public static void app() {  
    String s = "System.out.println(\"Memory Read StringReader Check\");  
    StringReader sIn = new StringReader(s);  
    int code;  
    try {  
        while ((code = sIn.read()) != -1) {  
            System.out.print((char) code + " ");  
        }  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

## Чтение из памяти и форматирование

- Чтение из памяти и форматирование DataInputStream class
  - DataInputStream работает только с потоком байт
  - поэтому нужна «прокладка» в виде потока байт ByteArrayInputStream (byte[] buffer)
- Работа с потоком байт
  - ByteArrayInputStream (byte[] buffer) чтение побайтно из массива байт
  - read() чтение одного байта
  - DataInputStream(<InputStrteam>) чтение данных разного типа из потока байт
  - read() чтение одного байта
  - readByte(), readShort(), readInt(), readLong(), readFloat(), readDouble() чтение в классы примитивов
  - readUnsignedByte(), readUnsignedShort() беззнаковые версии
  - readUTF() чтение в формате UTF8
  - readFully(byte[]) чтение всего потока данных в буфер
  - DataInputStream.available() определение конца потока
- **ВНИМАНИЕ.** поток работает с байтами, поэтому конец потока определяется методом available()
- Пример. реализация чтения в виде байт через InputStream [lesson\\_ch18/ch/ex12/classb](#)

```
public static void app() {  
    System.out.println("ByteArrayInputStream from String:");  
    String s = "System.out.println(\"Memory Read StringReader Check\");  
    ByteArrayInputStream in = new ByteArrayInputStream(s.getBytes()); //строку в байты  
    int code;  
    while (in.available() > 0) { // метод определения конца потока  
        code = in.read();  
        System.out.print((char) code + " ");  
    }  
    System.out.println("\nDataInputStream from String:");  
    DataInputStream dIn = new DataInputStream(new ByteArrayInputStream(s.getBytes()));  
    try {  
        while (in.available() != 0) {  
            code = dIn.read();  
            System.out.print((char) code + " ");  
        }  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

## Вывод в файл

- Ввод из файла
  - `StringReader(string)` чтение из строки и создает поток
  - `BufferedReader(stream)` чтение буферизованное из потока
- Вывод в файл
  - `FileWriter(file)` записывает данные в файл по `char, int, String, char[]`
  - `BufferedWriter` записывает данные намного быстрее `char, int, String, char[]`
  - `PrintWriter` записывает данные как форматированный вывод
  - `new PrintWriter(out)` вывод в файл через буферизованный вывод
  - `new PrintWriter(fileName)` вывод в файл через имя файла, вывод все равно буферизованный

- Пример. реализация вывода

```
public static void check(String fileName, String fileName2, String fileName3) {  
    try {  
        BufferedReader in = new BufferedReader(  
            new StringReader(BufferedInputStream.read(fileName)));  
        FileWriter fw = new FileWriter(fileName2);  
        BufferedWriter bw = new BufferedWriter(fw); // буферизованный вывод  
        PrintWriter pw = new PrintWriter(bw); // параметр буферизованный вывод  
        PrintWriter pf = new PrintWriter(fileName3); // параметр имя файла  
        int lineCount = 1;  
        int lineCount2 = 1;  
        String s;  
        while ((s = in.readLine()) != null) {  
            pw.println(lineCount++ + ":" + s);  
            pf.print(lineCount2++ + "." + s + "\n");  
        }  
        in.close();  
        pw.close();  
        pf.close();  
        System.out.println(BufferedInputStream.read(fileName2));  
        System.out.println(BufferedInputStream.read(fileName3));  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

- Форматированный вывод

```
public class BFileWrite {  
    public static void writeString(String fileName, String string) {  
        try {  
            PrintWriter pw = new PrintWriter(fileName);  
            pw.println(string);  
            pw.close();  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
    public static void writeList(List<String> list, String fileName) {  
        try {  
            FileWriter fw = new FileWriter(fileName);  
            BufferedWriter bw = new BufferedWriter(fw); // буферизованный вывод  
            PrintWriter pw = new PrintWriter(bw); // параметр буферизованный вывод  
            ListIterator<String> lit = list.listIterator(list.size());  
            int lineCount = list.size();  
            while (lit.hasPrevious()) {  
                pw.printf(Locale.ENGLISH, "%2d:%s\n", lineCount--, lit.previous());  
            }  
            pw.close();  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

## Сохранение и восстановление данных

- Сохранение и восстановление данных DataInputStream, DataOutputStream

### DataInputStream

- DataInputStream(<InputStrteam>) класс позволяет читать из потока данные в виде примитива
  - read() чтение одного байта
  - readByte(), readShort(), readInt(), readLong(), readFloat(), readDouble() чтение в классы примитивов
  - readUnsignedByte(), readUnsignedShort() беззнаковые версии
  - readUTF() чтение в формате UTF8
  - readFully(byte[]) чтение всего потока данных в буфер
  - DataInputStream.available() определение конца потока
- Потоки для DataInputStream
  - ByteArrayInputStream (byte[] buffer) поток байт данных
  - FileInputStream(fileName) поток данных с файла
  - read() читает байт данных
  - read(buff[]) читает в буфер до заполнения
- ВНИМАНИЕ.** Единственный надежный способ записать строку, чтобы потом прочитать это формат UTF8
- Методы readUTF(), writeUTF() ПОЗВОЛЯЮТ смешивать строки с другими типами данных

### DataOutputStream

- DataOutputStream класс позволяет записывать данные в виде примитива
  - те же функции только на запись
- Потоки для DataOutputStream
  - тоже самое только на вывод
- Пример. реализация ввода, вывода упаковки данных в примитивные типы [lesson\\_ch18/ch/ex15/codea](#)

### RandomAccessFile

- RandomAccessFile класс произвольного доступа к файлу
  - seek() функция произвольного доступа к файлу
  - read(), readByte(),,, readUTF() функции чтения, записи данных в формате примитивов
  - чтобы правильно работать с файлом, надо знать его структуру для позиционирования данных
- Пример. реализация доступа с seek() [lesson\\_ch18/ch/ex16/codea](#)
- ```
public class RandomAccess {
    private static String fileName = "./src/ch18/ex16/codea/RandomAccess.txt";
    public static void check() {
        try {
            new File(fileName).delete();
            RandomAccessFile rf = new RandomAccessFile(fileName, "rw");
            for (int i = 0; i < 7; i++) {
                rf.writeDouble(i*1.414);
            }
            rf.writeUTF("The end of the file");
            rf.close();
            display();
            rf = new RandomAccessFile(fileName, "rw");
            rf.seek(5*8);
            rf.writeDouble(47.0001);
            rf.close();
            display();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## TextFile Custom Class

- TextFile Class описание класса работы с текстовыми файлами
  - TextFile extends ArrayList<String> наследует ArrayList, конструктор строит сразу ArrayList
  - new File(fileName).getAbsoluteFile() использует абсолютный путь к файлу
  - finally{} используется для гарантированного закрытия файла
- **ВНИМАНИЕ.** используется вложенный try\_finally только для того, чтобы гарантированно закрыть файл

[lesson\\_ch18/ch/ex17/codea](#)

## Scanner

- Scanner класс чтения текстовых файлов
  - точно также как FileReader позволяет читать файлы, но не может их записывать
  - применяется для создания парсеров программного кода

## Чтение двоичных файлов

- Чтение двоичных файлов BufferedInputStream, FileInputStream
- FileInputStream(fileName) поток данных с файла
  - read() чтение байта
  - read(byte[]) чтение в буфер байт
  - available() количество доступных байт
- BufferedInputStream(stream)
  - read() чтение байта
  - read(byte[]) чтение в буфер байт
  - available() количество доступных байт
- Пример. реализация чтения байт из файла

[lesson\\_ch18/ch/ex19/codea](#)

## Ввод Вывод Буферизация Потоков

- Правильные способы буферизации
- Байт ориентированные потоки
  - InputStream, OutputStream
- Символ ориентированные потоки
  - Reader, Writer
- Output
  - PrintStream new PrintStream(new BufferedOutputStream(new FileOutputStream( filename )))
  - PrintWriter new PrintWriter(new BufferedWriter( new FileWriter( fileName )));
  -
- Input
  - BufferedInputStream new BufferedInputStream(new FileInputStream( filename ))
  - DataInputStream DataInputStream(new ByteArrayInputStream( byte[] ));
  - BufferedReader new BufferedReader(new InputStreamReader(new InputStream( stream ))))
  -

## Стандартный ввод вывод

- Стандартные потоки System.in, System.out и System.err
  - System.in стандартный поток ввода
  - System.out стандартный поток вывода
  - System.err стандартный поток вывод ошибок
- Для работы со стандартными потоками используются обертки
  - System.in BufferedReader(InputStreamReader) буферизация System.in
  - System.out
  - System.err

## System.In

- System.In входной поток, наследует низкоуровневый InputStream

[lesson\\_ch18/ch/ex21/codea](#)

- Пример. реализация ввода построчно System.in

```
public static void check() {  
    try {  
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));  
        String s;  
        while ((s = stdIn.readLine()) != null) {  
            System.out.println(s);  
        }  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

- Пример. реализация универсального транслятора данных In to Out

[lesson\\_ch18/ch/ex21/exercise](#)

- используются как стандартные потоки ввода вывода, так и файлы
  - метод flush() для принудительного вывода строки используется

## System.Out

- System.Out является объектом PrintStream, наследует OutputStream

- метод flush() вызывается автоматически autoflush = true, чтобы увидеть результат вывода

[lesson\\_ch18/ch/ex22/codea](#)

- Пример. реализация перевода System.Out в символьный поток

## Redirect Standard Stream

- Перенаправление Standard Stream
  - setIn(inputStream)
  - setOut(PrintStream)
  - setErr(PrintStream)
- **ВНИМАНИЕ.** потоки перед перенаправлением ОБЯЗАТЕЛЬНО сохранить в ссылках
- **ВНИМАНИЕ.** Как происходит разыменование после SetIn() SetOut() ??? почему нет in.close() ???
- Пример. реализация переключения потоков

[lesson\\_ch18/ch/ex22/codeb](#)

```
public static void check() {  
    String fileRead = "./src/ch18/ex22/codeb/RedirectStream.java";  
    String fileWrite = "./src/ch18/ex22/codeb/RedirectStream.txt";  
    PrintStream console = System.out; // просто ссылка на объект  
    try {  
        BufferedInputStream in = new BufferedInputStream(new  
                                         FileInputStream(fileRead));  
        PrintStream out = new PrintStream(new BufferedOutputStream(  
                                         new FileOutputStream(fileWrite)));  
        System.setIn(in); // переключить на новый поток из файла  
        System.setOut(out); // переключить на новый поток в файл  
        System.setErr(out); // ошибки тоже на новый поток в файл  
        // ВНИМАНИЕ. здесь уже действуют потоки клиента  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
        String s; //br это буферизованная версия потока клиента  
        while ((s = br.readLine()) != null) { // ввод из файла  
            System.out.println(s.toUpperCase()); // вывод в файл  
        }  
        out.close(); // закрыть поток вывода  
        System.setOut(console); // восстановление потока вывода системного  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

## Управление процессами

- Пример. реализация запуска процесса и управления потоками ввода вывода [lesson\\_ch18/ch/ex22/codec](#)

```
public static void command(String command) { // OS command to exec  
    boolean err = false;  
    try {  
        Process process = new ProcessBuilder(command.split(" ")).start(); // новый процесс  
        BufferedReader results = new BufferedReader(  
            new InputStreamReader(process.getInputStream()));  
        String s;  
        while ((s = results.readLine()) != null) { // данные с процесса выводить на out  
            System.out.println(s);  
        }  
        BufferedReader errors = new BufferedReader(  
            new InputStreamReader(process.getErrorStream()));  
        while ((s = errors.readLine()) != null) { // данные с процесса на err  
            System.err.println(s); //  
            err = true;  
        }  
    } catch (Exception e) {  
        if (command.toLowerCase().startsWith("cmd /c")) {  
            command("cmd /c " + command);  
        } else {  
            throw new RuntimeException(e);  
        }  
    }  
    if (err) { // была ошибка  
        throw new OSExecException("Errors executing " + command);  
    }  
}
```

## Новый ввод вывод nio

[lesson\\_ch18/ch/ex23/codea](#)

- Новый ввод вывод nio
- FileChannel файловый канал работающий с байтами поддерживает четыре класса
  - ByteBuffer это базовый класс обмена данными nio
  - FileInputStream класс ввода из файла
  - FileOutputStream класс вывода в файл
  - RandomAccess класс одновременно ввода и вывода файла
- **ВНИМАНИЕ.** При работе с ByteBuffer ОБЯЗАТЕЛЬНО вызывать метод flip() перед использованием byte[]
- FileChannel символьный не реализован, но есть методы которые поддерживают объекты
  - Reader чтение символьное
  - Writer запись символьная

## ByteBuffer

### • Методы ByteBuffer

- wrap() оборачивает массив byte[] в объект буфера
- put() запись примитивных типов
- allocate() резервирование памяти при создании объекта
- flip() обрезает буфер до текущей позиции и сбрасывает указатель на начало byte[]
- rewind() установить указатель на начало буфера
- clear() очищает буфер и устанавливает указатель в начало буфера
- read() чтение буфера
- mark() пометить позицию
- reset() вернуться к помеченной позиции
- position() получить текущую позицию
- position(int) задает текущую позицию
- capacity() емкость буфера
- limit() возвращает предельную позицию
- limit(int) задает предельную позицию
- remaining() возвращает разницу между текущей и предельной позицией
- hasRemaining() возвращает true если есть элементы между текущей и предельной позицией

### •

### • Пример. реализация FileChannel nio

[lesson\\_ch18/ch/ex23/codea](#)

```
public static void check(String fileRead, String fileWrite, String fileRead2) {  
    try {  
        FileChannel fc = new FileOutputStream(fileWrite).getChannel();  
        fc.write((ByteBuffer.wrap(TextFile.read(fileRead).getBytes()))); // целый файл  
        fc.close();  
        fc = new RandomAccessFile(fileWrite, "rw").getChannel(); // доступ чтение запись  
        fc.position(fc.size());  
        fc.write((ByteBuffer.wrap(TextFile.read(fileRead2).getBytes()))); // еще раз  
        fc.close();  
        fc = new FileInputStream(fileWrite).getChannel(); // на чтение и только  
        ByteBuffer bf = ByteBuffer.allocate(BSIZE);  
        fc.read(bf); // прочитать в буфер 1К данных  
        bf.flip(); // ОБЯЗАТЕЛЬНО подготовка к чтению  
        while (bf.hasRemaining()) { // есть данные  
            System.out.print((char)bf.get());  
        }  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

## FileChannel

- Классы
  - FileInputStream      класс ввода
  - FileOutputStream      класс вывода
  - RandomAccessFile      класс ввода/вывода

- Методы
  - getChannel()      получить канал доступа FileChannel

- Пример. реализация копирования вручную FileChannel

[lesson\\_ch18/ch/ex23/codeb](#)

```
public static void check(String fileSrc, String fileDst) {  
    final int BSIZE = 10;  
    try {  
        FileChannel fcIn = new FileInputStream(fileSrc).getChannel();  
        FileChannel fcOut = new FileOutputStream(fileDst).getChannel();  
  
        ByteBuffer buffer = ByteBuffer.allocate(BSIZE);  
        while (fcIn.read(buffer) != -1) { // читать до завершения файла либо размера  
            буфера  
            buffer.flip(); //  
            fcOut.write(buffer); // записать полученные данные  
            buffer.rewind(); // перевести назад в начало  
            fcOut.write(buffer); // повторно записать полученные данные  
            buffer.clear(); // очистить буфер данных  
        }  
  
        fcIn.close(); // почему не закрывает?  
        fcOut.close();  
  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

- Пример. реализация копирования автоматом FileChannel

[lesson\\_ch18/ch/ex23/codeb](#)

```
public static void check(String fileSrc, String fileDst) {  
    final int BSIZE = 10;  
    try {  
        FileChannel fcIn = new FileInputStream(fileSrc).getChannel();  
        FileChannel fcOut = new FileOutputStream(fileDst).getChannel();  
        fcIn.transferTo(0,fcIn.size(), fcOut); // первое копирование //pos src,0 dst  
        fcOut.transferFrom(fcIn, 10, fcIn.size()); // второе копирование //pos dst,0 src  
        fcIn.close();  
        fcOut.close();  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

- Применение

[lesson\\_ch18/ch/ex23/codeb](#)

```
public static void app() {  
    System.out.println("FileChannel manual copy:");  
    String fileIn = "./src/ch18/ex23/codeb/DataIn.txt";  
    String fileOut = "./src/ch18/ex23/codeb/DataOut.txt";  
    FileChCopy.check(fileIn,fileOut);  
  
    fileIn = "./src/ch18/ex23/codeb/Data2In.txt";  
    fileOut = "./src/ch18/ex23/codeb/Data2Out.txt";  
    FileChTran.check(fileIn,fileOut);  
}
```

## Преобразование строковых данных

- Способы перекодирования байтов в символы
  - `write(bf)` `read.Charset.forName().decode(bf)`
  - `write.getBytes(Encode)` `read.asCharBuffer()`
  - `write.asCharBuffer()` `read.asCharBuffer()`
- Пример. реализация указанных способов [lesson\\_ch18/ch/ex23/codec](#)
- ```
public static void check(String fileWrite) {
    try {
        FileChannel fc = new FileOutputStream(fileWrite).getChannel();
        fc.write(ByteBuffer.wrap(stringData).getBytes()); // строка получить байты
        fc.close();
        fc = new FileInputStream(fileWrite).getChannel(); // получить в виде потока
        ByteBuffer bf = ByteBuffer.allocate(BSIZE);
        // попытка вывода asCharBuffer()
        fc.read(bf); // прочитать всю строку с гарантией
        bf.flip(); // подготовить к чтению
        System.out.println("byte[] write.default() read.asCharBuffer():");
        System.out.println(bf.asCharBuffer()); // просто вывести выдает флаги
    // способ 1 with Charset.decode()
        bf.rewind(); // вернуть указатель на начало
        String charset = System.getProperty("file.encoding");
        System.out.println("\nbyte[] write() read.Charset(" + charset +").decode()");
        System.out.println(Charset.forName(charset).decode(bf));
    // способ 2 with write encoded(UTF-16) data
        fc = new FileOutputStream(fileWrite).getChannel(); // write channel
        fc.write(ByteBuffer.wrap(stringData).getBytes("UTF-16")); // write encoded
        fc.close();
        fc = new FileInputStream(fileWrite).getChannel(); // файл в виде потока
        bf.clear();
        fc.read(bf);
        bf.flip();
        System.out.println("\nbyte[] write.getbytes(UTF-16) read.asCharBuffer():");
        System.out.println(bf.asCharBuffer());
    // способ 3
        fc = new FileOutputStream(fileWrite).getChannel(); // write channel
        bf = ByteBuffer.allocate(stringData.length()*2); // размер в 2 раза
        bf.asCharBuffer().put(stringData); // записать через charBuffer
        fc.write(bf); // записать кодированные данные
        fc.close();
        fc = new FileInputStream(fileWrite).getChannel(); // файл в виде потока
        fc.read(bf);
        bf.flip();
        System.out.println("\nbyte[] write.asCharBuffer() read.asCharBuffer():");
        System.out.println(bf.asCharBuffer());
        fc.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```
- Применение [lesson\\_ch18/ch/ex23/codec](#)
- ```
public static void app() {
    BufferToText.check("./src/ch18/ex23/codec/CodeC.txt");
}
```
- Пример. реализация выдачи доступных CharSet в системе [lesson\\_ch18/ch/ex23/codec](#)

## Работа с обычными текстовыми файлами UTF8

- Работа с обычными текстовыми файлами UTF8
- **ВНИМАНИЕ.** DataInput/OutputStream НЕ РАБОТАЮТ с обычными файлами UTF8
- **ВНИМАНИЕ.** для обычных текстовых файлов использовать классы BufferedReader, BufferedWriter
- Пример. реализация работы с обычными файлами текста класс BFileParser [lesson\\_ch18/ch/ex23/codec](#)

## FileChannel и Примитивные типы

- FileChannel позволяет читать и писать байт данные в переменные примитивных типов
  - чтение и запись примитивов производятся через представление “as<Type>()”
  - методы представления as<Type>.put(), get<Type>() конвертируют примитивные типы в byte[]
- Пример. реализация работы FileChannel с примитивными типами

[lesson\\_ch18/ch/ex24/codea](#)

```
public static void check() {  
    final int BSIZE = 1024;  
    ByteBuffer bf = ByteBuffer.allocate(BSIZE);  
    bf.asCharBuffer().put("Hello World!"); //char  
    char ch;  
    System.out.print("char :");  
    while ((ch = bf.getChar()) != 0) {  
        System.out.print(ch + " ");  
    }  
    System.out.println();  
    bf.asShortBuffer().put((short) 2272); //short  
    System.out.println("short :" + bf.getShort());  
    bf.asIntBuffer().put(12272); //int  
    System.out.println("int :" + bf.getInt());  
    bf.asLongBuffer().put(112272); //long  
    System.out.println("long :" + bf.getLong());  
    bf.asFloatBuffer().put(1.2272F); //float  
    System.out.println("float :" + bf.getFloat());  
    bf.asDoubleBuffer().put(1.12272); //double  
    System.out.println("double:" + bf.getDouble());  
}
```

## Представление ByteBuffer

- Представление ByteBuffer
  - это применение классов примитивных типов совместимых с ByteBuffer
  - позволяет работать с byte[] как с массивом примитивного типа
- Методы <type>Buffer
  - flip() подрезать размер буфера до текущей позиции
  - rewind() указатель на начало буфера
  - position(int) задать новую позицию
  - get() получить данные с текущей позиции, указатель на следующую
  - put() записать данные в текущую позицию, указатель на следующую
  - get(n) прочитать данные с (n) позиции
  - put(n, val) записать данные в (n) позицию
- Пример. реализация работы с byte[] как с массивом int[]

[lesson\\_ch18/ch/ex24/codeb](#)

```
public static void check() {  
    final int BSIZE = 1024;  
    ByteBuffer bf = ByteBuffer.allocate(BSIZE);  
    IntBuffer ib = bf.asIntBuffer();  
    ib.put(new int[]{1, 5, 8, 10, 12, 14, 10, 47, 29, 34, 56, 79});  
    System.out.print("ib :");  
    show(ib);  
    ib.position(14); // это ключевое доступная позиция буфера  
    ib.put(256);  
    ib.put(32);  
    show(ib); //put()  
    System.out.println("ib.get(5) :" + ib.get(5)); //get()  
    ib.position(20); // новая текущая позиция  
    ib.flip(); // подрезает буфер по заданной позиции  
    show(ib); //flip()  
    ib.put(18, 305);  
    show(ib); //put(n, v)  
}
```

- Смена представления

[lesson\\_ch18/ch/ex24/codeb](#)

- один и тот же буфер byte[] можно считывать как буфер с разным форматом данных

Таблица размещения примитивов в byte[]

Type	Representation								Note
1 byte[]	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	
	0	0	0	0	0	0	0	97	
2 chars	[0]	[1]	[2]	[3]					
	" "	" "	" "	" "				"a"	
3 short	[0]	[1]	[2]	[3]					
	0	0	0	97					
4 int	[0]	[1]							
	0	97							
5 long	[0]								
	97								
6 float	[0]	[1]							
	0	1.36E-43							
7 double	[0]								
	4.8E-322								

- Пример. реализация представления данных в разных формах примитивов

[lesson\\_ch18/ch/ex24/codeb](#)

```
public static void check() {
    ByteBuffer bf = ByteBuffer.wrap(new byte[]{ 0, 0, 0, 0, 0, 0, 0, 0, 'a'});
    //byte
    bf.rewind();
    System.out.print(bf.position()+"->"+bf.get()+" , ");
    //char
    CharBuffer cb = ((ByteBuffer)bf.rewind()).asCharBuffer();
    System.out.print(cb.position()+"->"+cb.get()+" , ");
    //short
    ShortBuffer sb = ((ByteBuffer)bf.rewind()).asShortBuffer();
    System.out.print(sb.position()+"->"+sb.get()+" , ");
    //int
    IntBuffer ib = ((ByteBuffer)bf.rewind()).asIntBuffer();
    System.out.print(ib.position()+"->"+ib.get()+" , ");
    //long
    LongBuffer lb = ((ByteBuffer)bf.rewind()).asLongBuffer();
    System.out.print(lb.position()+"->"+lb.get()+" , ");
    //float
    FloatBuffer fb = ((ByteBuffer)bf.rewind()).asFloatBuffer();
    System.out.print(fb.position()+"->"+fb.get()+" , ");
    //double
    DoubleBuffer db = ((ByteBuffer)bf.rewind()).asDoubleBuffer();
    System.out.print(db.position()+"->"+db.get()+" , ");
}
```

## Выравнивание данных BigEndian LittleEndian

- Выравнивание данных BigEndian LittleEndian для класса ByteBuffer

- метод order(<endian>) задает тип выравнивания
- ByteOrder.BIG\_ENDIAN старший байт по адресу [0] младший [1] Java по умолчанию
- ByteOrder.LITTLE\_ENDIAN младший байт по адресу [0] старший [1]

- Пример. реализация проверки выравнивания через представление

[lesson\\_ch18/ch/ex25/codea](#)

```
public static void check() {
    bf.order(ByteOrder.BIG_ENDIAN);
    bf.putInt(16777216);
    System.out.println("int:"+ bf.getInt());
    bf.order(ByteOrder.LITTLE_ENDIAN);
    System.out.println("int:"+ bf.getInt());
}
```

## Буферы и манипуляция данными ByteBuffer

- Буферы и массивы примитивных типов

○ ByteBuffer	byte[]	array()	get(byte[])	wrap(byte[])
○ CharBuffer	char[]	array()	get(char[])	wrap(char[])
○ ShortBuffer	short[]	array()	get(short [])	wrap(short [])
○ IntBuffer	int[]	array()	get(int [])	wrap(int [])
○ LongBuffer	long[]	array()	get(long [])	wrap(long [])
○ FloatBuffer	float[]	array()	get(float [])	wrap(float [])
○ DoubleBuffer	double[]	array()	get(double [])	wrap(double [])

- Переход между ByteBuffer <Type>Buffer

○ Методы ByteBuffer	from ByteBuffer	to ByteBuffer
○ CharBuffer	asCharBuffer()	putChar()
○ ShortBuffer	as ShortBuffer ()	putShort ()
○ IntBuffer	as IntBuffer ()	putInt ()
○ LongBuffer	as LongBuffer ()	putLong ()
○ FloatBuffer	as FloatBuffer ()	putFloat ()
○ DoubleBuffer	as DoubleBuffer ()	putDouble ()

- **ВНИМАНИЕ.** переход ByteBuffer в другие БЫСТРЫЙ массивом, переход обратно МЕДЛЕННЫЙ по одному

- ByteBuffer и кодирование символов

○ Encoder	Charset.newEncoder()	encode(CharBuffer)	to ByteBuffer
○ Decoder	Charset.newDecoder()	decode(ByteBuffer)	to CharBuffer

- **ВНИМАНИЕ.** Какие бы манипуляции не производились с FileChannel все равно работает ByteBuffer

- Пример. реализация работы ByteBuffer и CharBuffer

[lesson\\_ch18/ch/ex25/codeb](#)

```
public class UsingBuffers {
    private static void scrambler(CharBuffer cb) {
        while (cb.hasRemaining()) {
            cb.mark(); // пометить позицию
            char c1 = cb.get();
            char c2 = cb.get();
            cb.reset(); // вернуться к отмеченной позиции
            cb.put(c2); // всегда возвращается к последней
            cb.put(c1); // помеченной позиции
        }
    }
    public static void check(String filename) {
        char[] charData = "UsingBuffers".toCharArray();
        ByteBuffer bb = ByteBuffer.allocate(charData.length * 2); // 2 байта на символ
        //CharBuffer cb = bb.asCharBuffer(); // это штатный способ для работы с файлами
        CharBuffer cb = CharBuffer.wrap(charData);
        System.out.println("cb initial : " + cb.rewind());
        scrambler(cb);
        System.out.println("cb scrambled:" + cb.rewind());
        try {
            FileChannel fc = new FileOutputStream(fileName).getChannel();
            bb = ByteBuffer.allocate(cb.length() * 2);
            for (cb.rewind(); cb.hasRemaining(); ) {
                bb.putChar(cb.get()); // перекачка по одному элементу
            }
            bb.flip();
            fc.write(bb);
            fc.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## Отображаемые в память файлы

- MappedByteBuffer      класс отображения файлов
  - используется для больших файлов, которые не умещаются в памяти
  - наследуется от ByteBuffer поэтому доступны все методы и представления ByteBuffer
- **ВНИМАНИЕ.** Для удаления отображаемого файла надо закрыть файл, освободить буфер, вызвать gc()
  - закрыть дескриптор FileChannel      fc.close()
  - освободить ссылку MappedBuffer      mbb = null
  - вызвать сборщик мусора      System.gc()
  - удалить отображаемый файл      new File(fileName).delete()
- Пример. реализация отображения файла 128Mb
- ```
public class LargeMapFile {  
    private static int length = 0x7FFFFFF; // 128Mb  
    public static void check() {  
        try {  
            System.out.println("File writing started:");  
            MappedByteBuffer mbb=new RandomAccessFile("./src/ch18/ex25/codec/test.dat","rw").  
                getChannel().map(FileChannel.MapMode.READ_WRITE, 0, length);  
            for (int i = 0; i < length ; i++) {  
                mbb.put((byte) 'x');  
            }  
            System.out.print("File data:");  
            for (int i = length/2; i < length/2+10; i++) {  
                System.out.print((char)mbb.get(i)+" ");  
            }  
            System.out.println();  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

## FileChannel чтение и запись стандартных текстовых файлов отображение

- Чтение из файла в строку или массив строк из FileChannel fc
  - отобразить весь файл в MappedByteBuffer или ByteBuffer bb = fc.map();
  - декодировать bb в CharBuffer cb = Charset.forName(System.getProperties("file.encodings").decode(bb))
  - получить весь файл в виде массива строк String s = cb.toString().split("\r\n");
- Запись строки или массива строк в из FileChannel fc
  - записать в StringBuilder, получить массив байт sb.toString().getBytes() записать fc.write(byte[])

## Производительность

- Производительность новой библиотеки nio с предыдущей версией
- Пример. реализация проверки производительности Stream и FileChannel
  - Stream      потоки
  - запись      DataOutputStream(BufferedOutputStream(FileOutputStream(file)))
  - чтение      DataInputStream(BufferedInputStream(FileInputStream(file)))
  - чтение,запись RandomAccessFile(file)
  - FileChannel      потоки
  - запись      RandomAccessFile(file).getChannel()
  - чтение      FileInputStream(file).getChannel()
  - чтение,запись RandomAccessFile(file).getChannel()
- **ВНИМАНИЕ.** FileChannel превосходит Stream по производительности по чтению, записи в 500 раз быстрее
- Пример. производительность ByteBuffer.allocate() и allocateDirect
- **ВНИМАНИЕ.** мощный пример классов тестирования

## Блокировка файлов FileChannel

- Блокировка файлов    FileLocking class
  - механизм синхронизации доступа к файлам между процессами
- Методы
  - lock()                                 блокирует весь файл, если недоступен то блокирует программу
  - lock(pos,size,bool shared)                тоже самое, но блокирует часть файла
  - tryLock()                                 блокирует весь файл, если недоступен продолжает программу
  - tryLock(pos,size,bool)                        тоже самое, но блокирует часть файла
  - release()                                 снять блокировку с файла
  - **ВНИМАНИЕ.** При увеличении файла блокировка всего файла действует на новый размер
  - **ВНИМАНИЕ.** При увеличении файла блокировка части файла действует только на старый участок

- Пример. реализация блокировки

[lesson\\_ch18/ch/ex27/codea](#)

```
public static void check() {
    String fileName = "./src/ch18/ex27/codea/TestData.txt";
    try {
        FileOutputStream fs = new FileOutputStream(fileName);
        FileLock fl = fs.getChannel().tryLock(); // попытка заблокировать файл на доступ
        if (fl != null) {
            System.out.println("File Locked");
            TimeUnit.MILLISECONDS.sleep(100);
            fl.release(); // освободить файл
            System.out.println("File unlocked");
        }
        fl.close();
        fs.close();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

## Блокировка файлов FileChannel Mapped Memory

- Блокировка файлов при отображении памяти
- Пример. реализация частичной блокировки честными потоками
  - открывается два потока, которые блокируют две разные части буфера
  - основной поток контролирует завершили они работу методом `isAlive()`
  - по завершении потоков, закрывается основной канал `fc.close()` и буфер отображения
- **ВНИМАНИЕ.** НЕ закрывать `FileChannel` пока потоки не отработали с буферами отображения

```
public class LockingMappedFiles {  
    static final int LENGTH = 0x7FFFFFF; //  
    static FileChannel fc;  
    private static String fileName = "./src/ch18/ex27/codea/test.dat"; // огромный файл  
    private static class LockAndModify extends Thread {  
        private ByteBuffer buff;  
        private int start, end;  
        public LockAndModify(ByteBuffer bb, int start, int end) {  
            this.start = start;  
            this.end = end;  
            bb.limit(end); // финальная позиция для внешнего буфера  
            bb.position(start); // текущая позицию внешнего буфера  
            this	buff = bb.slice(); //  
            start(); // запуск процесса  
        }  
        public void run() {  
            try {  
                FileLock fl = fc.lock(start, end, false);  
                System.out.println("Locked :" + start + " to " + end);  
                while (buff.position() < buff.limit() - 1) { // прокачать местный буффер  
                    buff.put((byte) (buff.get() + 1)); // заменить в локальном буфере  
                }  
                fl.release(); // разблокировать  
                System.out.println("\nReleased:" + start + " to " + end);  
                fl.close();  
                buff = null;  
            } catch (IOException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
    public static void check() {  
        try {  
            fc = new RandomAccessFile(fileName, "rw").getChannel();  
            System.out.print("Mapped Buffer filling ...");  
            MappedByteBuffer mb = fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);  
            for (int i = 0; i < LENGTH; i++) {  
                mb.put((byte) 'x');  
            }  
            System.out.println("done.");  
            LockAndModify p1 = new LockAndModify(mb, 0, 0 + LENGTH / 3); // 0..1/3 run()  
            LockAndModify p2 = new LockAndModify(mb, LENGTH / 2, LENGTH / 2 + LENGTH / 4);  
            while (p1.isAlive() || p2.isAlive()) {  
                Time.sleep(1);  
                System.out.print(".");  
            }  
            System.out.println("\nprocesses finished...");  
            fc.close(); // ВНИМАНИЕ НЕ ЗАКРЫВАТЬ БУФЕР ДО ЗАВЕРШЕНИЯ РАБОТЫ С НИМ  
            mb = null; // обязательно для удаления файла  
            System.gc(); // обязательно для удаления файла  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

lesson\_ch18/ch/ex27/codea

## Сжатие файлов

- Сжатие файлов
  - базируется на классах ввода вывода `InputStream`, `OutputStream`
  - библиотека Java работает по сжатию только с байтами
- **ВНИМАНИЕ.** GZIP сжимает поток, поэтому это всегда только ОДИН файл, обычно tar архив с файлами

## Подмена потоков

- Подмена потоков
  - для чтения потока как символьного из архива его можно обернуть в символьный поток
  - `symInStream` `BufferedReader(InputStreamReader( GZIPInputStreamReader(FileInputStream(file)))`
  - для записи потока символов в архив, его можно обернуть в символьный поток
  - `symOutStream` `BufferedReader(InputStreamReader( GZIPInputStreamReader(FileInputStream(file)))`
  -
- **ВНИМАНИЕ.** библиотека сжатия только с БАЙТАМИ для символов `InputStreamReader`, `OutputStreamWriter`
- **ВНИМАНИЕ.** GZIP архив работает так, что имя файла внутри равно имени архива без `.gz`
- поэтому для файла внутри архива `MyClass.java` архив должен называться `MyClass.java.gz`
- Методы сжатия

| Класс                             | Метод                      | Описание                                                                             |
|-----------------------------------|----------------------------|--------------------------------------------------------------------------------------|
| <code>CheckedInputStream</code>   | <code>getCheckSum()</code> | контрольная сумма для любого потока <code>InputStream</code> кроме потока распаковки |
| <code>CheckedOutputStream</code>  | <code>getCheckSum()</code> | контрольная сумма для любого потока <code>OutputStream</code> кроме потока сжатия    |
| <code>DeflaterOutputStream</code> |                            | базовый класс сжатия                                                                 |
| <code>ZipOutputStream</code>      |                            | сжимает в формате ZIP, наследует <code>DeflaterOutputStream</code>                   |
| <code>GZIPOutputStream</code>     |                            | сжимает в формате GZIP, наследует <code>DeflaterOutputStream</code>                  |
| <code>DeflaterInputStream</code>  |                            | базовый класс распаковки                                                             |
| <code>ZipInputStream</code>       |                            | распаковывает из ZIP, наследует <code>DeflaterOutputStream</code>                    |
| <code>GZIPInputStream</code>      |                            | распаковывает из GZIP, наследует <code>DeflaterOutputStream</code>                   |

-

## Простое сжатие в формате GZIP

- Простое сжатие в формате GZIP [lesson\\_ch18/ch/ex27/codeb](#)
- Работа с байтами  

```
br = new BufferedInputStream(new GZIPInputStream(new FileInputStream(fileOut)));
bw = new BufferedOutputStream(new GZIPOutputStream(new FileOutputStream(fileOut2));
```
- Работа с символами и строками  

```
br = new BufferedReader(new InputStreamReader( // читать GZIP без кодировки да
    new GZIPInputStream(new FileInputStream(fileOut)), "utf-8"));
bw = new BufferedWriter(new OutputStreamWriter(
    new GZIPOutputStream(new FileOutputStream(fileOut2)), "utf-8"));
```
- Пример. метод GZIPCompress.check() реализация работы с GZIP архивом [lesson\\_ch18/ch/ex27/codeb](#)
- **ВНИМАНИЕ.** приведены полный комплекс чтения записи с архивацией в GZIP
- **ВНИМАНИЕ.** GZIP это потоковый архиватор, в нем можно упаковать только один файл
- Как правило много файлов упаковывают в архив Tar и затем этот файл дожимают GZIP

## Многофайловый архив в формате ZIP

- Многофайловый архив в формате ZIP [lesson\\_ch18/ch/ex27/codeb](#)
- Работа с байтами  

```
FileOutputStream fw = new FileOutputStream(fileOut);
CheckedOutputStream cSumW = new CheckedOutputStream(fw, new Adler32());
ZipOutputStream zipW = new ZipOutputStream(cSumW);
BufferedOutputStream bw = new BufferedOutputStream(zipW); // байтовая запись
BufferedInputStream br = new BufferedInputStream(new FileInputStream(file));
```
- Работа с символами и строками  

```
FileOutputStream fw = new FileOutputStream(fileOut);
    CheckedOutputStream cSumW = new CheckedOutputStream(fw, new Adler32()); // 
ZipOutputStream zipW = new ZipOutputStream(cSumW);
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(zipW)); я запись
BufferedReader br = new BufferedReader(new FileReader(file));
```
- Пример. метод ZipCompress.check() реализация работы с ZIP архивом [lesson\\_ch18/ch/ex27/codeb](#)
- **ВНИМАНИЕ.** приведен полный комплекс чтения и записи с архивацией одно и нескольких файлов в ZIP

## Архивы JAR

- Архивы JAR

| Ключ   | Описание                                           | Примечание    |
|--------|----------------------------------------------------|---------------|
| c      | создает новый или пустой архив                     |               |
| t      | распечатать содержимое архива                      |               |
| x      | извлекает файлы                                    |               |
| x file | извлекает указанные файлы                          |               |
| f      | задает имя файла архива                            |               |
| m      | указывает что после имени файла идет имя манифеста |               |
| v      | выводит подробную информацию о выполнении          |               |
| 0      | указывает сохранять файлы в архив без сжатия       | для CLASSPATH |
| M      | не включать в манифест                             |               |

- Примеры команд
- jar -cf myJarFile.jar \*.class создать архив из файлов \*.class
- jar -cmf myJarFile.jar manifestFile.mf \*.class создать архив и туда же свой манифест
- jar -tf myJarFile.jar показать содержимое архива
- jar -tvf myJarFile.jar показать содержимое и диагностику
- jar -cvf myJarFile.jar folder1 folder2 создать архив из folder1,folder2
- **ВНИМАНИЕ.** если указана папка, добавляются все файлы и каталоги внутри папки
- **ВНИМАНИЕ.** если файл создан с ключом -0 то он может быть указан в CLASSPATH как библиотека классов

## Serialization сериализация объектов

- Сериализация объектов
  - запись объектов программы на диск называется <сериализацией>
  - применяется для использования объектов в течение нескольких пусков остановов программы
  - применим к любому классу который поддерживает интерфейс Serializable
- Условия сериализации
  - класс Serializable должен иметь конструктор по умолчанию
  - если базовый класс Serializable, то потомки также поддерживают Serializable
  - static fields non - Serializable by default
  - **ВНИМАНИЕ.** для сохранения static полей использовать static методы [lesson\\_ch18/ch/ex30/coded](#)
  - transient fields non - Serializable by default
  - **ВНИМАНИЕ.** для сохранения transient использовать Externalizable [lesson\\_ch18/ch/ex30/codeb](#)
  - non-Serializable подкласс может быть восстановлен при условии что
  - подкласс должен иметь конструктор без аргументов и он доступен из Serializable класса
  - поля подкласса будут инициализированы через public или protected конструктор
- Назначение сериализации
  - для поддержки удаленного вызова методов RMI когда аргументы транспортируются по сети
  - для поддержки визуальных компонентов JavaBean для восстановления визуальных объектов
- **ВНИМАНИЕ.** Библиотека Hibernate является мощным механизмом сериализации объектов.
- Процедура сериализации объекта
  - при сериализации объекта сохраняются все ссылки на другие объекты и сами объекты
  - то есть практически сохраняется вся среда, «паутина» объектов
- Запись объекта
  - создается выходной поток OutputStream, который обворачивается в ObjectOutputStream
  - метод writeObject() сериализует объект и передает его в выходной поток OutputStream
- Чтение объекта
  - создается поток InputStream, который обворачивается в ObjectInputStream
  - метод readObject() восстанавливает объект и возвращает ссылку Object
  - исходящий кастинг полностью воссоздает объект нужного класса
- Пример. реализация сериализации объектов на диск или массив [lesson\\_ch18/ch/ex27/codec](#)

## Serialization и Class restore

- Поиск и восстановление класса [lesson\\_ch18/ch/ex28/codea](#)
  - если объект прибыл по сети, и неизвестен его класс, встает задача поиска класса объекта
  - при чтении объекта возможно определить его класс
- **ВНИМАНИЕ.** если класс объекта не находится, то выдается Exception

## Управление Serialization

- Управление Serialization создание дополнений и расширений в Serialization
  - создать класс объекта, который реализует интерфейс Externalizable
  - реализовать методы интерфейса
- Пример. реализация интерфейса Externalizable [lesson\\_ch18/ch/ex28/codeb](#)
- Необходимые условия работы с объектами Externalizable
  - обязательно должен быть PUBLIC конструктор по умолчанию без параметров
  - инициализация полей в другом конструкторе с параметрами и только в нем
  - явная запись и чтение полей в соответствующих методах writeExternal() и readExternal()

## Transient ключевое слово

- Transient ключевое слово
  - используется для выборочной отмены сериализации
- Методы выборочной сериализации
  - реализовать интерфейс Externalizable
  - использовать ключевое слово transient
- Пример. реализация выборочной сериализации

[lesson\\_ch18/ch/ex30/codea](#)

и прописать в методах свою процедуру  
для выборочной отмены сериализации для полей

[lesson\\_ch18/ch/ex30/codea](#)

## Альтернатива Externalizable

- Альтернатива Externalizable
  - применить интерфейс Serializable
  - добавить метод writeObject() обязательно вот такого вида

```
private void writeObject(ObjectOutputStream stream) throws IOException {}
```
  - добавить метод readObject() обязательно вот такого вида

```
private void readObject(ObjectInputStream stream) throws IOException,  
ClassNotFoundException {}
```
  - добавить в методы свои команды сохранения и восстановления данных
- **ВНИМАНИЕ.** Методы writeObject() readObject заменят оригинальные и отработают задачи пользователя
- Методы writeDefaultObject() и readDefaultObject отрабатывают Transient по умолчанию
- Пример. реализация своего интерфейса

[lesson\\_ch18/ch/ex30/codeb](#)

```
public class SerialCtl implements Serializable {  
    private String a;  
    private transient String b;  
    public SerialCtl() {  
    }  
    public SerialCtl(String a, String b) {  
        this.a = "Non transient:"+a;  
        this.b = "Transient      :" +b;  
    }  
    private void writeObject(ObjectOutputStream stream) throws IOException {  
        stream.defaultWriteObject();  
        stream.writeObject(b);  
    }  
    private void readObject(ObjectInputStream stream) throws IOException,  
                           ClassNotFoundException {  
        stream.defaultReadObject();  
        b =(String)stream.readObject();  
    }  
    @Override  
    public String toString() {  
        return a + "\n" + b;  
    }  
}
```

## Долговременное хранение

- Долговременное хранение
  - это хранение объектов данных на диске для последующего восстановления состояния программы
- Правила работы со временем хранения объектов
  - всегда работать в одном потоке для сохранения контекста программы
  - статические поля восстанавливать вручную через `serializeStaticState()`, `deserializeStaticState()`
  - не использовать код чтения классов **объектов <????>**
  -
- **ВНИМАНИЕ.** Для корректного сохранения объектов делать все в ОДНОМ потоке
- Пример. Работа в одном потоке и проблемы с восстановлением static полей [lesson\\_ch18/ch/ex30/coded](#)
  - запускать отдельно файл `StoreCADSystem` и `RestoreCADSystem`
  - так как классы хранятся в памяти и выгрузить их невозможно, только запускать из разных `main()`
- Пример. Правильная работа с статическими полями [lesson\\_ch18/ch/ex30/exercise](#)
- ```
public class LineS extends Shapes {
    private static int color = RED;
    public LineS(int xPos, int yPos, int dimension) {
        super(xPos, yPos, dimension);
    }
    @Override
    public void setColor(int color) {
        this.color = color;
    }
    @Override
    public int getColor() {
        return color;
    }
    public static void serializeStaticState(ObjectOutputStream stream)
        throws IOException{
        stream.writeInt(color);
    }
    public static void deserializeStaticState(ObjectInputStream stream)
        throws IOException{
        color = stream.readInt();
    }
}
```
- Применение. Сохранение
- ```
public static void check() {
    String fileOut = "./src/ch18/ex30/exercise/state.dat";
    try {
        ObjectOutputStream sOut = new ObjectOutputStream(
            new FileOutputStream(fileOut)); //store static
        LineS.serializeStaticState(sOut); // метод сохранения Line класс
        sOut.writeObject(shapes); // весь список сохранить на диск

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```
- Применение. Восстановление
- ```
public static void check() {
    try {
        ObjectInputStream sIn = new ObjectInputStream(
            new FileInputStream(fileIn)); //restore static
        LineS.deserializeStaticState(sIn); // вызвать метод восстановления Line
        List<Shapes> shapes = (List<Shapes>) sIn.readObject(); // список объектов
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

## XML работа с форматом

- XML работа с форматом
  - Библиотеки работы с XML
  - java.xml      входит в состав JDK
  - XOM            бесплатная библиотека
- XML формат
  - тэг            текст скобках означает группу текста            <тэг>    текст    </тэг>
  -
- XML функции
  - разбор            разбор текста для получения структуры документа
  - поиск            поиск данных по структуре данных текста
  - проверка        проверка данных по структуре текста
  - преобразование    преобразование данных в структуре текста
  -

[lesson\\_ch18/ch/ex31/codea](#)

## Parsing

- Parsing анализ текста
  - разбор текста и выделение текста по признакам на структуру
  - сохранение частей текста в определенную структуру DOM <Document Object Model>
- DOM структура
  - node    узел документа который связан со своим тэгом
  - tree     дерево документа набор узлов объединенных в единую структуру

## Классы работы с XML

- DocumentBuilder        класс создания XML документа
- Document            класс собственно XML документа
  - getDocumentElement()    получить корневой узел документа
- Node                класс тэга или узла документа
  - getNodeName()        получить узел из документа
  - getNodeType()        получить тип узла      Node.TEXT\_NODE      просто #text его пропускать
  - getTextContent()    получить содержимое текста узла
  - getChildNodes()     получить список дочерних узлов NodeList<>
- NodeList            класс списка узлов
  - item(n)            получить узел из списка NodeList
  - getLength()        получить длину списка
- Пример. реализация работы с DOM

[lesson\\_ch18/ch/ex31/codea](#)

## Редактирование XML

- Редактирование XML
  - добавление в документ XML данных с новыми значениями
  - сохранение измененного документа в файл
- Пример. реализация создания и чтения файлов XML
- Пример. реализация создания и чтения файлов XML
- Пример. реализация ArrayList и чтение запись в XML
- Пример. реализация ArrayList и чтение запись в XML
- Пример. реализация Map и чтение и запись в XML
- 

[lesson\\_ch18/ch/ex31/codeb](#)

[lesson\\_ch18/ch/ex31/codeb](#)

[lesson\\_ch18/ch/ex31/codec](#)

[lesson\\_ch18/ch/ex31/coded](#)

[lesson\\_ch18/ch/ex31/exercise](#)

[lesson\\_ch18/ch/ex32/exercise](#)

## Preferences API

- Preferences API
  - структура как Map, состоит из полей <key:value>
  - позволяет сохранять примитивы и строки, длина строки не может быть более 8кБайт
  - предназначена для сохранения настроек на диске
  - работают с системой компа, сохраняют информацию для пользователя
- Методы
  - Preferences.userNodeForPackage(Preferences.class);                                  сохранить в контексте пользователя
  - Preferences.systemNodeForPackage(Preferences.class);                                  сохранить в контексте системы
  - В качестве узла используется класс
- **ВНИМАНИЕ.** для non-static методов в качестве параметра вызывается getClass()
- **ВНИМАНИЕ.** вся информация предпочтений хранится в РЕЕСТРЕ
- Пример. реализация Preferences API

[lesson\\_ch18/ch/ex33/codea](#)

```
public class PreferencesDemo {  
    public static void check() {  
        try {  
            System.out.println("If Windows issue Warning");  
            System.out.println("Goto HKLM/Software/JavaSoft >> make new Key Prefs");  
// это узел хранения пользователя  
            Preferences prefs = Preferences.userNodeForPackage(Preferences.class);  
//  
            Preferences prefs = Preferences.systemNodeForPackage(Preferences.class);  
            prefs.put("Location", "Oz");  
            prefs.put("FootWear", "Ruby Slippers");  
            prefs.putInt("Companions", 4);  
            prefs.putBoolean("Are the witches?", true);  
            int usageCount = prefs.getInt("UsageCount", 0);  
            usageCount++;  
            prefs.putInt("UsageCount", usageCount);  
            for (String key : prefs.keys()) {  
                System.out.printf("%-28s:%s\n", key, prefs.get(key, null));  
            }  
            System.out.println("How many companions :" + prefs.getInt("Companions", 0));  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

- Пример. реализация Preferences привязанные к каталогу

```
public class PrefCatalog {  
    public static void check(String pathName) {  
        try {  
            System.out.println("If Windows issue Warning");  
            System.out.println("Goto HKLM/Software/JavaSoft >> make new Key Prefs");  
            System.out.println("\nPreference Demo:");  
            Preferences prefs = Preferences.userRoot().node( // привязан к каталогу  
                new File(pathName).getAbsolutePath()); // это узел хранения пользователя  
            int pathCount = prefs.getInt("pathCount", 0);  
            pathCount++;  
            prefs.putInt("pathCount", pathCount);  
            for (String key : prefs.keys()) {  
                System.out.printf("%-28s:%s\n", key, prefs.get(key, null));  
            }  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

- 

[lesson\\_ch18/ch/ex33/exercise](#)

## Enumerations

### Enumerations Перечисления

- Enumerations Перечисления
  - класс констант, используется для создания массивов значений
- Методы
  - values() массив констант в порядке их объявления
  - ordinal() порядковый номер в массиве
  - compareTo() результат сравнения <int>
  - name() имя перечисления какое задано в классе Enum<>

[lesson\\_ch19/ch/ex01/codea](#)

### Staitc import Enum

- Статический импорт Enum
  - static import <path>.EnumClass.\* директива которая позволяет работать напрямую
- Пример.

```
public enum Spiciness {  
    NOT, MILD, MEDIUM, HOT, FLAMING  
}  
  
public class Burrito {  
    Spiciness degree;  
    public Burrito(Spiciness degree) {  
        this.degree = degree;  
    }  
    @Override  
    public String toString() {  
        return "Burrito is " + degree;  
    }  
}
```
- Применение.

```
import static ch19.ex01.codeb.Spiciness.*;  
public class CodeB {  
    public static void app() {  
        System.out.println("import static ch19.ex01.codeb.Spiciness.*");  
        System.out.println(new Burrito(NOT));  
        System.out.println(new Burrito(MEDIUM));  
        System.out.println(new Burrito(HOT));  
    }  
}
```

[lesson\\_ch19/ch/ex01/codeb](#)

## Добавление методов к Enum

- Добавление методов к Enum
  - перечисление работает как обычный класс, но Enum НЕ поддерживает НАСЛЕДОВАНИЕ
  - можно добавлять поля и методы в класс перечисления
- ВНИМАНИЕ. Обязательно должен быть private конструктор на точный набор полей Enum
- Пример. реализация полей и методов, ТРИ конструктора ОБЯЗАТЕЛЬНЫ
- ```
public enum OzWitch {  
    WEST("Miss Gulch, aka the Wicked Witch of the West"),  
    NORTH("Glinda, the Good Witch of the Nprth"),  
    EAST("Wicked Witch of the East, wearer of the Ruby " +  
        "Slippers, cruched by Dorothy's house"),  
    SOUTH("Good by inference, but missing"),  
    // тоже работает любая комбинация аргументов  
    WEST_SOUTH(12),  
    NORTH_WEST(15),  
    EAST_SOUTH("East winds transformations", 12),  
    EAST_WEST("Combine factors of influence", 15) ;  
    private String description; // без конструктора выдает ошибку  
    private int value;  
    // ВНИМАНИЕ конструкторы по умолчанию private  
    OzWitch(String description, int value) { // для объектов EAST_SOUTH, EAST_WEST  
        this.description = description;  
        this.value = value;  
    }  
    OzWitch(int value) { // для объектов WEST_SOUTH, NORTH_WEST  
        this.value = value;  
    }  
    OzWitch(String description) { // для объектов WEST,NORTH,EAST,SOUTH  
        this.description = description;  
    }  
    public String getDescription() {  
        return description+"."+value;  
    }  
}
```
- Применение.
- ```
public static void app() {  
    for (OzWitch ozWitch : OzWitch.values()) {  
        System.out.println(ozWitch+": "+ozWitch.getDescription());  
    }  
}
```

[lesson\\_ch19/ch/ex01/codec](#)

[lesson\\_ch19/ch/ex01/codec](#)

## Переопределение методов Enum

- Переопределение методов Enum
  - внутри объявления класса можно переопределять базовые методы, например `toString()`
- Пример. реализация переопределения `toString()`
- ```
public enum SpaceShip {  
    SCOUT, CARGO, TRANSPORT, CRUISER, BATTLESHIP, MOTHERSHIP;  
    @Override  
    public String toString() {  
        char chStart = name().charAt(0);  
        String lower = name().substring(1, name().length()-1);  
        char chEnd = name().charAt(name().length()-1);  
        return chStart+lower.toLowerCase()+chEnd;  
    }  
}
```

[lesson\\_ch19/ch/ex01/codec](#)

## Enum and Switch

- Enum and Switch
  - Enum специально создавались для работы в конструкции switch{}
  - при использовании Enum компилятор не отслеживает секцию default:
- **ВНИМАНИЕ.** Всегда отслеживать секцию default при Enum and Switch
- Пример. реализация Enum and Switch
- ```
public class TrafficLight {  
    Signal color = Signal.RED;  
    public void change() {  
        switch (color) {  
            case RED:  
                color = Signal.GREEN;  
                break;  
            case GREEN:  
                color = Signal.YELLOW;  
                break;  
            case YELLOW:  
                color = Signal.RED;  
                break;  
            default:  
                break;  
        }  
    }  
    @Override  
    public String toString() {  
        return "The traffic light is " + color;  
    }  
}
```

[lesson\\_ch19/ch/ex01/coded](#)

[lesson\\_ch19/ch/ex01/coded](#)

[lesson\\_ch19/ch/ex01/coded](#)

## Методы Enum

- Метод Enum.values()
  - класс Enum содержит свои базовые методы
  - компилятор добавляет статический метод values()
- Доступ к значениям Enum без метода values()
  - при восходящем преобразовании к Enum values() недоступен, так как добавляется компилятором
  - метод Class.getEnumConstants() позволяет получить доступ к полям Enum
- Пример. реализация доступа к полям без метода values()
- ```
public enum Search {  
    HITTER, YON  
}
```
- ```
public class UpCastEnum {  
    public static void check() {  
        Search[] vals = Search.values();  
        Enum e = Search.HITTER; // объект назначен по восходящему  
        // System.out.println(e.values()); // не работает  
        Enum[] eVals = e.getClass().getEnumConstants(); // работает  
    }  
}
```
- **ВНИМАНИЕ.** Метод getEnumConstants() работает для ЛЮБОГО класса, но всем кроме Enum выдает NULL
- Пример. реализация доступа к полям класса не являющегося Enum

[lesson\\_ch19/ch/ex02/codea](#)

[lesson\\_ch19/ch/ex02/codeb](#)

[lesson\\_ch19/ch/ex02/codeb](#)

## Enum и Интерфейсы реализация

- Реализация Enum через интерфейсы
  - Enum невозможно наследовать
  - Enum можно создать реализации нескольких интерфейсов
  - Enum позволяет внутри размешать интерфейсы с другими Enum внутри
- Пример. реализация интерфейса IGenerator
  - стандартный механизм реализации интерфейса, как с обычными классами

[lesson\\_ch19/ch/ex02/codec](#)

[lesson\\_ch19/ch/ex02/codec](#)

## Реализация Enum и Generic

- Случайный выбор
  - реализация универсального генератора по Enum<>
- Пример. реализация работы с Enum как с объектом класса для генерации
- ```
public enum Activity {
    SITTING, LYING, STANDING, HOPPING, RUNNING, DODGING, JUMPING, FALLING, FLYING
}
```
- ```
public class Enums {
    private static Random rnd = new Random();
    public static <T extends Enum> T random(Class<T> eClass) {
        T[] values = eClass.getEnumConstants();
        return values[rnd.nextInt(values.length)];
    }
}
```

[lesson\\_ch19/ch/ex03/codea](#)

## Enum и Интерфейсы организация кода

- Enum и Интерфейсы организация кода
  - интерфейс может включать несколько Enum, которые реализуют ДАННЫЙ интерфейс
- **ВНИМАНИЕ.** Используется ЗАЦИКЛИВАНИЕ интерфейса на Enum которое реализует этот интерфейс
- Порядок инициализации Enum с параметрами
  - для каждой константы вызывается конструктор
  - конструктор создает значение или ряд значений
- Описание реализации Enum

[lesson\\_ch19/ch/ex03/codeb](#)

[lesson\\_ch19/ch/ex03/codec](#)

```
public enum Course {
    APPETIZER(Food.Appetizer.class), // при переборе values вызывается конструктор
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] foods;
    Course(Class<? extends Food> kind) { // конструктор заполняет значение values
        foods = kind.getEnumConstants();
    }
    public Food randomSelection() {
        return Enums.random(foods); // через массив
    }
}
```

- Применение

```
public static void check() {
    for (int i = 0; i < 5 ; i++) { // перебирает пять наборов еды
        for (Course course : Course.values()) { // перебирает все константы класса
            Food food = course.randomSelection(); // для каждой генерится свое foods[]
            System.out.println(food); // генерится случайное блюдо
        }
    }
}
```

## Компактное решение

- Компактное решение
  - разместить интерфейсы внутри объекта Enum
- **ВНИМАНИЕ.** Enum константы ДОЛЖНЫ быть сразу же за объявлением класса Enum
- ```
public enum SecurityCategory {  
    STOCK(Security.Stock.class), // объекты SecurityCategory  
    BOND(Security.Bond.class); // аргументы класса Security Interface  
    private interface Security {  
        enum Stock implements Security {  
            SHORT, LONG, MARGIN;  
        }  
        enum Bond implements Security {  
            MUNICIPAL, JUNK;  
        }  
    }  
    Security[] securities; // заполняется при обращении к объектам  
    SecurityCategory(Class<? extends Security> kind) {  
        securities = kind.getEnumConstants();  
    }  
    public Security randomSelection() {  
        return Enums.random(securities); // выбрать случайное значение  
    }  
  
    public static void check() {  
        System.out.println("Enum and Interface Compact:");  
        for (int i = 0; i < 10; i++) { // выбрать из двух значений BOND:STOCK  
            SecurityCategory category = Enums.random(SecurityCategory.class);  
            System.out.println(category+" : "+category.randomSelection()); // из 3:2  
        }  
    }  
}
```

[lesson\\_ch19/ch/ex03/coded](#)

## Enum циклизмение класса

- Enum циклизмение класса
  - каждая константа класса Enum является объектом этого класса
  - все поля и методы класса доступны объектам этого класса общее правило
  - все поля и методы объекта УНИКАЛЬНЫ для каждой КОНСТАНТЫ которые объекты класса Enum
- Пример. реализация класса Enum и значения value уникального для констант [lesson\\_ch19/ch/ex03/codeb](#)
  - пример называется CheckEnum
  - константы класса INTER, DAKOM, LENAR, PASTE по сути объекты этого класса
  - константы INTER, DAKOM, LENAR, PASTE связаны с value, которое уникально для каждого из объектов
- ```
public enum CheckEnum {  
    INTER, DAKOM, LENAR, PASTE;  
    private int value;  
    public int getValue() {  
        return value;  
    }  
    public void setValue(int value) {  
        this.value = value;  
    }  
}
```
- Применение.
- ```
public static void check() {  
    for (CheckEnum checkEnum : CheckEnum.values()) { // задать значения  
        checkEnum.setValue(Range.getInt(100));  
    }  
    for (CheckEnum checkEnum : CheckEnum.values()) { // получить значения  
        System.out.println(checkEnum.name()+" "+checkEnum.getValue());  
    }  
}
```

[lesson\\_ch19/ch/ex03/codeb](#)

## EnumSet для хранения флагов

- EnumSet для хранения флагов
  - это набор Set для хранения битовых флагов
  - во внутреннем представлении это один или несколько типов long (64 бита)
- Пример. реализация короткого и длинного набора EnumSet

```
public enum AlarmPoints {  
    STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM, UTILITY, KITCHEN  
}  
  
public static void check() {  
    EnumSet<AlarmPoints> points = EnumSet.noneOf(AlarmPoints.class); //пустой набор класса  
    points.add(BATHROOM); // used static import  
    System.out.println("add : "+points);  
    points.addAll(EnumSet.of(STAIR1, STAIR2, KITCHEN)); // набор через addAll  
    System.out.println("addAll EnumSet.of : "+points);  
    points.addAll(EnumSet.allOf(AlarmPoints.class)); // можно добавить но смысла нет  
    System.out.println("addAll EnumSet.allOf : "+points);  
    points = EnumSet.allOf(AlarmPoints.class); // добавить весь класс  
    System.out.println("= EnumSet.allOf : "+points);  
    points.removeAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));  
    System.out.println("removeAll EnumSet.of : "+points);  
    points.removeAll(EnumSet.range(OFFICE1, OFFICE4));  
    System.out.println("removeAll EnumSet.range: "+points);  
    points = EnumSet.complementOf(points); // отрабатывает полный набор  
    System.out.println("EnumSet.complementOf : "+points); // и получает недостающие  
}
```

[lesson\\_ch19/ch/ex07/codea](#)

## EnumMap использование

- EnumMap использование
  - работает очень быстро за счет final и потому что базируется на массиве
  - размер фиксирован добавление или удаление ключей невозможно
  - put() работает только для существующих ключей
  - позволяет заменять методы отработки ситуаций
- EnumMap with EnumSet Keys
  - порядок следования ключей зависит строго от порядка включения в EnumSet<>
- **ВНИМАНИЕ.** Возможно заменять методы значений по ходу программы
- Пример. реализация работы EnumMap с разными методами в поле значений [lesson\\_ch19/ch/ex08/codea](#)

```
public interface ICommand {  
    void action();  
}  
  
public static void check() {  
    EnumMap<AlarmPoints, ICommand> cMap = new EnumMap<>(AlarmPoints.class);  
    cMap.put(KITCHEN, new ICommand() {  
        public void action() {  
            System.out.println("Kitchen Light On");  
        }  
    });  
    cMap.put(UTILITY, new ICommand() {  
        public void action() {  
            System.out.println("Tools Locked Off");  
        }  
    });  
    cMap.get(UTILITY).action();  
    cMap.put(UTILITY, new ICommand() {  
        public void action() {  
            System.out.println("Tools Started Working");  
        }  
    });  
    cMap.get(UTILITY).action();  
}
```

[lesson\\_ch19/ch/ex08/codea](#)

## Constant specific methods

- Методы привязанные к объектам константам Enum
  - присвоение объекту своего метода поведения
  - каждой константе в Enum присваивается свой метод поведения
- ВНИМАНИЕ. В отличие от EnumMap здесь методы фиксированы и не меняются по ходу программы
- Пример. реализация AlarmPoints но с постоянными методами

```
public enum AlarmConstMethods implements ICommand {  
    KITCHEN{  
        @Override  
        public void action() {  
            System.out.println("Kitchen Light On");  
        }  
    },  
    BATHROOM {  
        @Override  
        public void action() {  
            System.out.println("Bathroom Water On");  
        }  
    },  
    UTILITY {  
        @Override  
        public void action() {  
            System.out.println("Tools Locked Off");  
        }  
    };  
    public static void check() { // это статический метод общий для всех  
        for (AlarmConstMethods constMethod : values()) {  
            System.out.println(constMethod.name() + " : ");  
            constMethod.action();  
        }  
    }  
}
```

[lesson\\_ch19/ch/ex08/codeb](#)  
[lesson\\_ch19/ch/ex08/coded](#)  
[lesson\\_ch19/ch/ex08/coded](#)

- Проверка Enum на полиморфность

- объекты Enum это анонимный класс, и не может выступать как класс для создания объектов
- каждый Enum : очень похож на внутренний класс
- : не может обращаться к нестатическим полям
- : не может обращаться к методам внешнего класса

- Пример. реализация Enum с разными методами

```
public enum LakeClass {  
    WINKEN {  
        void behavior() {  
            System.out.println("LakeClass.WINKEN");  
        }  
    },  
    BLINKEN {  
        void behavior() {  
            System.out.println("LakeClass.BLINKEN");  
        }  
    }  
}  
• Проверка на тип данных:  
• public class NotClass {  
    private class Int {};  
    // void f1(LakeClass.WINKEN node) { // не работает, объект создать нельзя  
    // }  
    public static void check() {  
        System.out.println("WINKEN class :" + LakeClass.WINKEN.getClass());  
        System.out.println("BLINKEN class:" + LakeClass.BLINKEN.getClass());  
    }  
}
```

## Методы констант Переопределение

- Методы констант Переопределение
  - если все объекты используют один и тот же метод
  - возможно переопределить общий метод для конкретного экземпляра
- **ВНИМАНИЕ.** Переопределение работает в зависимости от доступа
  - default, public override работает и метод переопределяется
  - private override НЕ работает и метод класса перебивает метод объекта констант
- Пример. реализация переопределения
- ```
public enum OverrideConstantSpecific {  
    NUT, BOLT,  
    WASHER {  
        void f() {  
            // работает если доступ default или public  
            System.out.println("Overridden method"); // метод локальный объекта константы  
        }  
    };  
    void f() {  
        System.out.println("Default method"); // метод класса  
    }  
    public static void check() {  
        for (OverrideConstantSpecific ocs : OverrideConstantSpecific.values()) {  
            System.out.print(ocs.name() + " : ");  
            ocs.f();  
        }  
    }  
}
```

[lesson\\_ch19/ch/ex08/coded](#)

[lesson\\_ch19/ch/ex08/codee](#)

## «Chain of Responsibility» Pattern

- «Chain of Responsibility» Pattern
  - в этом паттерне поступивший запрос передается по цепочке пока не будет отработан
  - суть метода в том, чтобы объекту сопоставить несколько enum
  - для каждого типа enum запустить специфическую обработку
- Пример. реализация паттерна <Chain of Responsibility>
- ```
object mail обладает 5 переменными Enum deliver, scan, read, addr, return  
обработчик содержит 5 констант обработки, название не имеет значение  
каждая константа обработки имеет один и тот же метод Enum Constant Method  
каждый констант метод работает только со своим полем из deliver, scan, read, addr, return  
начальное назначение свойств идет по enum.random()  
обработчик перебирает по методу values() свои константы с методами обработки
```
- **ВНИМАНИЕ.** мощная реализация
- Пример. реализация паттерна <Chain of Responsibility> на базе enumMap
- **ВНИМАНИЕ.** мощная реализация

[lesson\\_ch19/ch/ex08/codee](#)

[lesson\\_ch19/ch/ex08/codef](#)

[lesson\\_ch19/ch/ex08/def](#)

[lesson\\_ch19/ch/ex09/exercise](#)

## Конечные автоматы

- Конечные автоматы
  - это структуры которые обладают конечным числом состояний
  - Enum идеально подходят для конечных автоматов, так как имеют фиксированный набор констант
- Пример. реализация КА на базе enum VendingMachine [lesson\\_ch19/ch/ex10/codea](#)
- Пример. мощная реализация VMachine масштабируется, загрузка из текста [lesson\\_ch19/ch/ex11/exercise](#)

## Множественная диспетчеризация

- Множественная диспетчеризация
  - в JAVA конструкция a.plus(b) или a.multiple(b) где a,b типы Generic <A,B> НЕ работает
  - для решения такой проблемы применяется множественная диспетчеризация
- Пример. организация работы с несколькими типами через интерфейс [lesson\\_ch19/ch/ex12/codea](#)
- ```
public interface IItem {
    Outcome compete(IItem i); // используется собственно интерфейс
    Outcome eval(Papers p); // класс реализующий интерфейс IItem
    Outcome eval(Scissors s); // класс реализующий интерфейс IItem
    Outcome eval(Rock r); // класс реализующий интерфейс IItem

}

public class Papers implements IItem { // отработка взаимодействия с каждым новым типом
    public Outcome compete(IItem i) { // прилетает неизвестный тип один из трех
        return i.eval(this); // все типы работают по IItem
    }
    public Outcome eval(Papers p) { // метод отрабатывает тип Paper
        return DRAW;
    }
    public Outcome eval(Scissors s) { // метод отрабатывает тип Scissors
        return WIN;
    }
    public Outcome eval(Rock r) { // метод отрабатывает тип Rock
        return LOSE;
    }
}
```
- Применение. работа с универсальным типом
  - public class RoShamBo1 {  
    private static Random rnd = new Random();  
    private static IItem newItem() {  
        switch (rnd.nextInt(3)) {  
            default:  
            case 0:  
                return new Scissors();  
            case 1:  
                return new Papers();  
            case 2:  
                return new Rock();  
        }  
    }  
    private static void match(IItem a, IItem b) {  
        System.out.printf("%s\n", a.compete(b)); // отработка универсальная типов  
    }  
    public static void check() {  
        for (int i = 0; i < SIZE; i++) {  
            match(newItem(), newItem()); // генерация объектов разных типов  
        }  
    }  
}
  - 
  - 
  -

## Диспетчеризация с использованием Enum

- Диспетчеризация с использованием Enum
  - интерфейсы работают с классами, Enum объекты не являются классами
  - для Enum используется механизм switch\_case
- Пример. реализация усложненного вианта сEnum

```
public interface ICompetitor<T extends ICompetitor<T>> { // расширяет ICompetitor<T>
    Outcome compete(T item);
}

public enum Outcome {
    WIN, LOSE, DRAW // выиграл, проиграл, ничья
}

public class RoShambo {
    public static <T extends ICompetitor<T>> void match (T a, T b) { // под интерфейс
        System.out.printf("%-10s vs %-10s : %s\n", a, b, a.compete(b)); // RoShamBo2<>
    }
    public static <T extends Enum<T> & ICompetitor<T>>
        void play(Class<T> rsbClass, int size) {
        for (int i = 0; i < size; i++) { // ext Enum<T> for Enums, ICompetitor<> for match
            match(Enum.random(rsbClass), Enums.random(rsbClass));
        }
    }
}

public enum RoShamBo2 implements ICompetitor<RoShamBo2> {
    PAPER(DRAW, LOSE, WIN),
    SCISSORS(WIN, DRAW, LOSE),
    ROCK(LOSE, WIN, DRAW);
    private static Random rnd = new Random();
    private Outcome vPaper, vScissors, vRock;
    RoShamBo2(Outcome vPaper, Outcome vScissors, Outcome vRock) {
        this.vPaper = vPaper;
        this.vScissors = vScissors;
        this.vRock = vRock;
    }

    @Override
    public Outcome compete(RoShamBo2 item) {
        switch (item) {
            default:
            case PAPER:
                return vPaper;
            case SCISSORS:
                return vScissors;
            case ROCK:
                return vRock;
        }
    }

    private static RoShamBo2 newItem() {
        return values()[rnd.nextInt(values().length)];
    }
    private static void match(RoShamBo2 a, RoShamBo2 b) {
        System.out.printf("%-10s vs %-10s : %s\n", a, b, a.compete(b));
    }

    public static void check() {
        System.out.println("Standard Variant:");
        RoShambo.play(RoShamBo2.class, 20);

        System.out.println("\nHome variant:");
        for (int i = 0; i < 20; i++) {
            match(newItem(), newItem());
        }
    }
}
```

lesson\_ch19/ch/ex12/codeb

## Использование методов констант Enum

- Использование методов констант Enum
  - для обработки используется перечисление,
  - каждый пункт перечисления содержит свой метод с switch, осуществляющий диспетчеризацию
- Пример. реализация на базе Enum и методов констант [lesson\\_ch19/ch/ex12/codec](#)

## Диспетчеризация на базе EnumMap

- Диспетчеризация на базе EnumMap
  - используется двойное EnumMap
  - базовая EnumMap перебирает объекты, дочерние варианты для каждого объекта
- Пример. реализация диспетчеризации на базе EnumMap [lesson\\_ch19/ch/ex12/coded](#)
- ```
public enum RoShamBo5 implements ICompetitor<RoShamBo5>{
    PAPER, SCISSORS, ROCK;
    static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>> map = new
        EnumMap<RoShamBo5.class); // ключи взяли из Enum
    static {
        for (RoShamBo5 roShamBo5 : RoShamBo5.values()) { // прогнать PAPER,SCISSORS, ROCK
            map.put(roShamBo5, new EnumMap<RoShamBo5, Outcome>(RoShamBo5.class));
        }
        initRow(PAPER, DRAW, LOSE, WIN);
        initRow(SCISSORS, WIN, DRAW, LOSE);
        initRow(ROCK, LOSE, WIN, DRAW);
    }
    private static void initRow(RoShamBo5 item, Outcome vPaper,
        Outcome vScissors, Outcome vRock) {
        EnumMap<RoShamBo5,Outcome> rMap = map.get(item); // получить карту выбора
        rMap.put(PAPER,vPaper);
        rMap.put(SCISSORS,vScissors);
        rMap.put(ROCK,vRock);
    }
    @Override
    public Outcome compete(RoShamBo5 item) { // двойная диспетчеризация
        return map.get(this).get(item); // вытаскиваем карту this из нее значение item
    }
    public static void check() {
        RoShambo.play(RoShamBo5.class,20);
    }
}
```

## Диспетчеризация на базе двумерного массива Enum

- Диспетчеризация на базе двумерного массива Enum
  - на базе Enum строится двумерный массив
  - решение производится просто выбором
- Пример. реализация диспетчеризации на базе Enum [lesson\\_ch19/ch/ex12/coded](#)

# Annotation

## Annotation Аннотации

- Annotation это метаданные в составе кода программы
  - аннотация состоит из знака @ плюс ключевое слово
  - аннотации используются также как модификаторы (cast)
- Аннотации общего назначения
  - @Override определение метода переопределяет базовый класс или интерфейс
  - @ Deprecated компилятор должен выдать предупреждение при использовании этого элемента
  - @SupressWarning подавляет неподходящие предупреждения компилятора

## Базовый синтаксис

- Базовый синтаксис
  - аннотации создаются отдельным файлом, как интерфейс
  - аннотации требуют определения @Target и @Retention
  - аннотации объявляются как файл класса с конструкцией @interface
- Пример. аннотации, объявление полей int, String, область действия метод [lesson\\_ch20/ch/ex01/codea](#)
- ```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description" ; // значение есть
}
public class Testable {
    @UseCase (id = 21)           // обязательно задать только что нет по умолчанию, id()
    void execute() {             // располагать только над методами
        System.out.println("Testable.execute");
    }
}
```

[lesson\\_ch20/ch/ex01/codea](#)
- Аннотация без элементов « маркерная аннотация» @Test [lesson\\_ch20/ch/ex01/codea](#)
- Аннотация отслеживания сценариев @UseCase [lesson\\_ch20/ch/ex01/codea](#)
- Пример. объявление поля Class, область действия класс [lesson\\_ch20/ch/ex01/codeb](#)
- ```
Target(ElementType.TYPE)                      // to class only
@Retention(RetentionPolicy.RUNTIME)            // at runtime only
public @interface SimulatingNull {
    public int id() default -1;
    public String description() default "";
    public long count() ;
    public Class value() default Ball.class;
}
```

[lesson\\_ch20/ch/ex01/codeb](#)
- Пример. объявление поля Annotation, @Constraints аннотация, область действия поле класса [lesson\\_ch20/ch/ex01/codea](#)  

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
    int value() default 0;
    String name() default "";
    Constraints constraints() default @Constraints; // объявление аннотаций
}
```
- Пример. задание значений в вложенной аннотации [lesson\\_ch20/ch/ex01/codeb](#)
- ```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Uniqueness {
    Constraints constraints() default @Constraints(unique = true);
}
```

## Мета аннотации

- Таблица Мета аннотации

Ключ	Поле	Описание
@Target	ElementType	область действия
	CONSTRUCTOR	объявление конструктора
	FIELD	объявление поля, и перечислений
	LOCAL_VARIABLE	объявление переменной
	METHOD	объявление метода
	PACKAGE	объявление пакета
	PARAMETER	объявление параметра
	TYPE	объявление класса, интерфейса, аннотации, перечисления
@Retention	RetentionPolicy	продолжительность хранения информации аргументы
	SOURCE	аннотации игнорируются компилятором
	CLASS	аннотации доступны в *.class, но могут игнорироваться JVM
	RUNTIME	аннотации сохраняются в JVM доступны через Reflections
@Documented	Javadoc	аннотация включается в Javadoc
@Inherited	Subclass access	Subclass могут наследовать аннотации базовых классов

## Написание обработчиков аннотаций

- Написание обработчиков аннотаций [lesson\\_ch20/ch/ex01/codea](#)
  - обработчики ищут в классе объекты, методы или поля, извлекают связанные аннотации
  - после фильтрации аннотации выбранного класса обрабатываются
- Пример. реализация обработчика аннотаций UseCaseTracker [lesson\\_ch20/ch/ex01/codea](#)
- ```

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description" ;
}

public class UseCaseTracker {
    private static void trackUseCases(List<Integer> useCases, Class<?> className) {
        for (Method method : className.getDeclaredMethods()) {
            UseCase uc = method.getAnnotation(UseCase.class); // получить аннотацию
            if (uc != null) { // данного типа для метода
                System.out.println("Found UseCase:" + uc.id() + " " + uc.description());
                useCases.remove(new Integer(uc.id())); // удалить из списка отработанные
            }
        }
        for (int useCase : useCases) {
            System.out.println("Warning: Missing use case-" + useCase);
        }
    }
    public static void check() {
        List<Integer> useCases = new ArrayList<>(Arrays.asList(47, 48, 49, 50));
        trackUseCases(useCases, PasswordUtils.class);
    }
}

```

## Элементы аннотаций

- Элементы аннотаций
  - это то, что может быть внутри класса аннотаций, то есть поля и методы
- Перечень допустимых элементов
  - все примитивные типы
  - String
  - Class            объекты типов классов
  - enum            объекты класса наследника Enum
  - Annotation      вложенный класс аннотации
  - массивы всех перечисленных типов

## Ограничение значений по умолчанию

- Ограничение значений по умолчанию
  - элементы аннотации должны иметь значение, либо по умолчанию, либо назначенное в объявления
  -
- **ВНИМАНИЕ.** В аннотации все элементы должны иметь значение либо по умолчанию, либо назначенное.
- **ВНИМАНИЕ.** Элементы не примитивных типов НЕ получают null по умолчанию
- Пример. реализация аннотации с значениями которые заменяют null [lesson\\_ch20/ch/ex01/codeb](#)

## Аннотации и наследование

- **ВНИМАНИЕ.** Аннотации не поддерживают наследование

## Генерирование внешних файлов

- Генерирование внешних файлов [lesson\\_ch20/ch/ex01/codec](#)
  - аннотации позволяют создавать внешние файлы с информацией об исходном коде
- Пример. реализация сообщения обработчика аннотаций для SQL [lesson\\_ch20/ch/ex01/codec](#)
  - @DBTable аннотация напоминание, что надо создать базу данных
  - @Constraints аннотация вложенная, задает параметры поля SQL, можно переназначить
  - @SQLString аннотация для полей String
  - @SQLInteger аннотация для полей Integer
  - @Uniqueless аннотация, показывает как задать новые параметры Constraints прямо в аннотации
  - **ВНИМАНИЕ.** Показано всего два поля в целях демо, по идее надо объявить все поля таблицы SQL
- Пример. реализация дополнительно полей пользователя с Enum [lesson\\_ch20/ch/ex01/exercise](#)

## Использование Apt для обработки аннотаций JDK7 max

- **ВНИМАНИЕ.** Утилита apt устарела, библиотеки sun.mirror не рекомендованы к применению
- Полное описание как работать смотреть в Приложение А
- **ВНИМАНИЕ.** Утилита apt доступна только в JDK7, далее только
- Apt это утилита для обработки аннотаций, работает с исходными \*.java файлами
  - apt при запуске требует файл класса фабрики, либо путь до классов фабрик
  - apt обрабатывает исходники пока во время обработки создаются новые файлы исходников
  - затем apt компилирует все файлы проекта
- **ВНИМАНИЕ.** При запуске apt надо указать путь до классов фабрик, иначе будет запущен сложный поиск
- AnnotationProcessorFactory
  - это интерфейс, на базе которого работает apt
  - для каждой аннотации нужен свой обработчик
- Пример. реализация аннотации которая извлекает открытые методы класса [lesson\\_ch20/ch/ex02/codea](#)

## Использование Javac для обработки аннотаций

- Использование Javac для обработки аннотаций
  - работа с javac производится точно также как и с apt из командной строки
  - для обработки нужен исходный файл \*.java, файл процессора \*.class, пути сохранения результата
  - отладки не получится, так как запуск программы идет с командной строки cmd ./c javac .....
    - либо нужно подключить через проект maven удаленный debugger
    - либо работать через выводы программы сообщений на консоль
- Особенности работы Javac
  - работает со своей библиотекой, поэтому javac и классы javaх отличаются от apt и sun.mirror
  - запуск процесса аннотаций производится в три этапа
    - системный вызов из программы пользователя с использованием ProcessBuilder
    - запуск System Commander Windows и вызов bat файла
    - запуск javac с параметрами из bat файла
- **ВНИМАНИЕ.** Чтобы вызвать несколько команд в одну строку использовать <&> «cmd /c cd src & dir»
- Пример. обработчик аннотаций на базе javac [lesson\\_ch20/ch/ex02m/codea](#)
- Пример. обработчик аннотаций на базе apt [lesson\\_ch20/ch/ex02/codea\](#)

## Использование «Visitor» Pattern

- Pattern «Visitor»
  - общий смысл в том, что создается отдельный класс Visitor методов обработки данных клиента
  - затем перебирается коллекция клиентов, и каждый вызывает accept(Visitor.pre, Visitor.post)
  - Visitor.pre      класс методов обработки данных клиента начальные
  - Visotor.post     класс методов обработки данных завершающие
- Применение к аннотациям
  - Annotation Processor перебирает элементы класса и вызывает метод accept(Visitor.pre, NO\_OP)
  - Visitor.pre      класс обработки всех элементов класса
  - NO\_OP            пустой элемент, заглушка
  - Обработка        accept отдает объекту Visitor элемент и для всего внутри вызывается свой метод  
◦ обработчик интерфейса DeclarationVisitor класса SimpleDeclarationVisitor
- Пример. реализация обработчика Visitor вариант javac СХЕМАТИЧНО [lesson\\_ch20/ch/ex03m/codea](#)

```
@SupportedAnnotationTypes({
    "dbase.DBTable", "dbase.Constraints", "dbase.SQLString", "dbase.SQLInteger"
})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class TCPFactory extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment env) {
        for (Element element : env.getRootElements()) { // перебор классов
            element.accept(new V1(), null); // элемент самого класса
            for (Element element1 : element.getEnclosedElements()) {
                element1.accept(new V1(), null); // перебор элементов внутри класса
            }
        }
        return true;
    }
    private class V1 extends SimpleElementVisitor8<Void, Void> {
        public Void visitVariable(VariableElement e, Void aVoid) {
            return super.visitVariable(e, aVoid); // обработка поля класса
        }
        public Void visitType(TypeElement e, Void aVoid) {
            return super.visitType(e, aVoid); // обработка элемента класса
        }
    }
}
```

## Модульное тестирование AtUnit @Unit

[lesson\\_ch20/ch/ex04/codea](#)

- Модульное тестирование @Unit
  - это framework написанный автором для тестирования кода автоматом при помощи аннотаций
- Теория
  - подключается библиотека @Unit
  - @Test аннотация теста, устанавливается перед методом тестируемого класса
  - аннотация требует чтобы метод не получал аргументов и возвращал void или boolean
  - return true      метод прошел тест
  - return false     метод не прошел тест
  - AtUnit.main(args)    запуск тестирования, args[] массив путей до файлов классов от <user dir>
- Особенности работы @Unit
  - для каждого теста @Unit
  - создается объект тестируемого класса через default constructor
  - @TestObjectCreate    объект инициализируется статическим методом
  - @TestObjectCleanup    объект освобождает ресурсы статическим методом
  - @TestProperty        объявляет метод временным, после тестирования метод удаляется
  - проводится тест с вызовом тестируемого метода
  - объект тестируемого класса уничтожается

• **ВНИМАНИЕ.** Порядок вызова методов НЕИЗВЕСТЕН тест не должен зависеть от порядка вызова методов

• Пример.

[lesson\\_ch20/ch/ex04/codea](#)

```
public class AtUnitExample1 {  
    public String methodOne() {  
        return "This is methodOne";  
    }  
    public int methodTwo() {  
        System.out.println("This is methodTwo");  
        return 2;  
    }  
    @Test // тестирование метода 1  
    boolean methodOneTest() { // тест метод не имеет аргументов возвращает bool или void  
        return methodOne().equals("This is methodOne");  
    }  
    @Test // тестирование метода 2  
    boolean m2() { // тест метод не имеет аргументов возвращает bool или void  
        return (methodTwo() == 2);  
    }  
    @Test  
    boolean failureTest() { // Shows output for failure  
        return false;  
    }  
    @Test  
    boolean anotherDissappointment() { // Shows output for failure  
        return false;  
    }  
    public static void check() { // процедура тестирования  
        try {  
            // указать полный путь до файла *.class от <user_dir>  
            String[] args = new String[]{  
                "out\\production\\jb01\\ch20\\ex04\\codea\\AtUnitExample1" };  
            AtUnit.main(args);  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

## Наследование и Композиция для тестирования методов класса

- Наследование как способ тестирования
  - позволяет тестировать методы без внедрения аннотаций @Test в код тестируемого класса
- Пример. реализация тестирования через наследования @Unit [lesson\\_ch20/ch/ex04/codeb](#)
- Композиция как способ тестирования
  - объект класса тестирования включается в состав полей класса тестируемого
- Пример. реализация тестирования через композицию @Unit [lesson\\_ch20/ch/ex04/codeb](#)

## Использование assert в @Unit

- Использование assert в @Unit
  - применение assert в библиотеке @Unit сделано автоматом
  - при ошибке или исключении assert выдается как неудавшийся тест
- **ВНИМАНИЕ.** Assert и AtUnit.main() должны находиться в РАЗНЫХ классах
- Пример. реализация assert в @Unit [lesson\\_ch20/ch/ex04/codec](#)
  - assert взводится и выдает сообщение только при ошибке, при ошибке тест провален
  - exception вылетает автоматом, сообщение выдается системой, при ошибке тест провален
- Тестирование класса пользователя
  - создается обычный класс пользователя с объектом
  - в каждый метод встраивается проверка работы объекта с assert :message
- Пример. реализация assert для простого объекта HashSet [lesson\\_ch20/ch/ex04/coded](#)
- Пример. реализация assert для HashSet тестирование и наследование [lesson\\_ch20/ch/ex04/coded](#)
- Пример. реализация assert для LinkedList тестирование [lesson\\_ch20/ch/ex06/exercise](#)
- Пример. реализация assert для LinkedList тестирование и наследование [lesson\\_ch20/ch/ex07/exercise](#)

## Инициализация объекта тестируемого класса

- Инициализация объекта тестируемого класса
  - когда требуется сложное конструирование и default constructor не подходит
  - создается статический метод и присоединяется к нему аннотация @TestObjectCreate
- Пример. реализация инициализации объекта тестируемого класса [lesson\\_ch20/ch/ex08/codea](#)
- ```
public class AtUnitExample3 {  
    private int n;  
    public AtUnitExample3(int n) {  
        this.n = n;  
    }  
    public String methodOne() {  
        return "This is methodOne";  
    }  
    @TestObjectCreate  
    public static AtUnitExample3 create() {  
        return new AtUnitExample3(15);  
    }  
    @Test  
    boolean initialization() {  
        return n == 15;  
    }  
  
    @Test  
    boolean methodOneTest() {  
        return methodOne().equals("This is methodOne");  
    }  
}
```

## Использование тестовых полей объекта тестируемого класса

- Использование тестовых полей объекта тестируемого класса
  - создаются дополнительные поля, только для целей тестирования
  - к каждому такому полю подсоединяется аннотация TestProperty
- Пример. реализация тестирования @Unit с тестовыми полями
- **ВНИМАНИЕ.** Порядок вызова методов НЕИЗВЕСТЕН тест не должен зависеть от порядка вызова методов
- Поэтому результаты отличаются от тех что в книге

```
public class AtUnitExample4 {  
    public static String theory = "All brontosaurus are thin at one end";  
    private String word;  
    private Random rnd = new Random();  
    public AtUnitExample4(String word) {  
        this.word = word;  
    }  
    public String getWord() {  
        return word;  
    }  
    public String scrambledWord() {  
        List<Character> chars = new ArrayList<>();  
        for (char c : word.toCharArray()) {  
            chars.add(c);  
        }  
        Collections.shuffle(chars, rnd); // перемешали в списке все символы  
        StringBuilder sb = new StringBuilder();  
        for (Character aChar : chars) {  
            sb.append(aChar);  
        }  
        return sb.toString();  
    }  
    @TestProperty // статические поля равные для всех экземпляров класса  
    public static List<String> input = Arrays.asList(theory.split(" ")); //разбить слова  
    @TestProperty  
    public static Iterator<String> words = input.iterator();  
    @TestObjectCreate  
    public static AtUnitExample4 createForTest() { // проверка другого имени  
        if (words.hasNext()) {  
            return new AtUnitExample4(words.next());  
        } else {  
            return null;  
        }  
    }  
    @Test  
    boolean words() { // этот метод вызывается первый и берет первое слово  
        System.out.print(" "+getWord()+" ");  
        return getWord().equals("All"); // отработает только на слово "are"  
    }  
    @Test  
    public boolean scramble1() { // этот метод вызывается третий и берет третье слово  
        rnd = new Random(15); // фиксированное seed для сравнения  
        System.out.print(" "+getWord()+" ");  
        String scrambled = scrambledWord();  
        System.out.println(scrambled);  
        return scrambled.equals("era");  
    }  
    @Test  
    public boolean scramble2() { // этот метод вызывается второй и берет второе слово  
        rnd = new Random(125); // фиксированное seed  
        System.out.print(" "+getWord()+" ");  
        String scrambled = scrambledWord();  
        System.out.println(scrambled);  
        return scrambled.equals("rsouasrotnsube");  
    }  
}
```

[lesson\\_ch20/ch/ex08/codeb](#)

- Использование методов завершения теста
- Использование методов завершения теста
  - `@TestObjectCleanup` аннотация используется завершения когда тесты все прошли
  - подключить ее к методу, который закрывает объект
- Особенности применения
  - `@TestObjectCreate` работает только с статическим методом
  - `@TestObjectCleanup` работает только с статическим методом
  - `@TestProperty` объявляет метод временным, после тестирования метод удаляется
- **ВНИМАНИЕ.** Так как `Create` и `Cleanup` статические методы, то поля с которыми они работают статические
- Пример. реализация закрытия файлов тестируемого объекта

[lesson\\_ch20/ch/ex08/codec](#)

```

public class AtUnitExample5 {
    private String text;

    public AtUnitExample5(String text) {
        this.text = text;
    }

    @TestProperty
    public static PrintWriter output;
    @TestProperty
    public static int counter;

    @TestObjectCreate
    public static AtUnitExample5 create() { // проверка другого имени
        String id = Integer.toString(counter++);
        try {
            output = new PrintWriter("src/ch20/ex08/codec/Test" + id + ".txt");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return new AtUnitExample5(id); // создается файл при создании каждого объекта
    }
    @TestObjectCleanup
    public static void cleanup(AtUnitExample5 obj) {
        System.out.println("Running cleanup "+ obj+ " "); // obj проинициализирован
        obj.output.close();
    }

    @TestProperty
    private int n = 15;

    @Test
    boolean test1() {
        output.print(text + ": test1 "+n++);
        return true;
    }
    @Test
    boolean test2() {
        output.print(this + ": test2 "+n++);
        return true;
    }
    @Test
    boolean test3() {
        output.print(text + ": test3 "+n++);
        return true;
    }
    @Override
    public String toString() {
        return text;
    }
}

```

## Использование @Unit с Generic

- Использование @Unit с Generic
  - Проблема: Тесты в принципе не могут работать с Generic
  - Решение: Тестирование проводится с классом который наследует Generic
- Пример. реализация @Unit и Класс<T> и наследованием в Класс <String> [lesson\\_ch20/ch/ex08/coded](#)
- **ВНИМАНИЕ.** Добавлен временный метод get() доступа к private полю
- Тестируемый базовый класс с Generic<T>
- ```
public class StackL<T> {  
    private LinkedList<T> linkedList = new LinkedList<T>();  
    @TestProperty  
    protected LinkedList<T> get() { // получить закрытый список  
        return linkedList;  
    }  
    public void push(T v) {  
        linkedList.addFirst(v);  
    }  
    public T pop() {  
        return linkedList.removeFirst();  
    }  
    public T top() {  
        return linkedList.getFirst();  
    }  
}
```
- Тестирующий класс потомок с типом <String>

```
public class StackLStringTest<String> extends StackL {  
    @Test  
    void _push() {  
        push("one");  
        assert top().equals("one"); // проверяем функцию  
        push("two");  
        assert top().equals("two"); // проверяем функцию  
    }  
    @Test  
    void _pop() {  
        push("one");  
        push("two");  
        assert pop().equals("two"); // проверяем функцию  
        assert pop().equals("one"); // проверяем функцию  
    }  
    @Test  
    void _top() {  
        push("A");  
        push("B");  
        assert top().equals("B"); // проверяем функцию  
        assert top().equals("B"); // проверяем функцию  
    }  
    @Test  
    void _list() { // вытаскивает private list из базового класса времененным методом  
        push("one");  
        push("two");  
        System.out.println(get().toString());  
    }  
}
```

## Реализация @Unit

- @Unit состоит из головного файла AtUnit.java и сопутствующих файлов аннотаций
- Пример. реализация новой аннотации @TestNote вывода диагностики [lesson\\_ch20/ch/ex11/exercise](#)
  - создана копия AtUnit.java файл AtUnitM.java, класс обработчик Proc.java, аннотация @TestNote
  - анализ и вывод аннотаций производит утилита Proc.checkClass(), procCheckField()
  - анализ и вывод аннотаций для методов встроен в программу AtUnitM.java process()

# Concurrency

## Многопоточное Программирование

- Многопоточное программирование
  - применяется для ускорения вычислений
  - в Java используется потоковая модель с вытеснением
  - в Java используется один процесс который разбивается на несколько потоков
  - каждый поток это отдельная независимо выполняемая программа

## Определение задач

- Объект Runnable [lesson\\_ch21/ch/ex01/codea](#)
  - это задача на выполнение, чтобы задача заработала ее надо прикрепить к потоку выполнения
  - по умолчанию задача прикрепляется к тому же потоку что и метод main(){}
  - интерфейс Runnable содержит только метод run(), который выполняется в бесконечном цикле
  - для потокового режима надо задачу присоединить к своему потоку при помощи класса Thread
- Пример. реализация объекта задача с интерфейсом Runnable [lesson\\_ch21/ch/ex01/codea](#)

## Класс Thread

- Класс Thread [lesson\\_ch21/ch/ex02/exercise](#)
  - это класс который преобразует объект Runnable в выполняемую задачу
- Процедура присоединения задачи к потоку
  - создать объект Thread(поток) с аргументом конструктора Объект Runnable
  - запустить метод Thread().start()
- Пример. реализация запуска 10 потоков с задачей генератора Fibonacci [lesson\\_ch21/ch/ex02/exercise](#)
  - реализован генератор чисел с предустановкой
- ```
public class FbRun implements Runnable {
    private static int count = 0;
    private final int id = count++;

    private int n;
    public FbRun(int n) {
        this.n = n;
    }

    @Override
    public void run() {
        Fibonacci f = new Fibonacci();
        for (int i = 0; i < n; i++) {
            System.out.print("#"+id+"."+f.next()+" ");
        }
        System.out.println();
        Thread.yield(); // освобождаем ресурсы
    }
}
```
- Присоединение задач к потокам
- ```
public class FbRunThreads {
    public static void check() {
        Random rnd = new Random();
        for (int i = 0; i < 10; i++) {
            new Thread(new FbRun(rnd.nextInt(10)+2)).start();
        }
    }
}
```

## Использование Executor



## Возвращение значений из задач

- Возвращение значений из задач
    - для получения возврата используется задача с интерфейсом Callable<T>
  - Методы ExecutorService
    - submit() собственное запуск задачи и подсоединение к потоку
    -
  - **ВНИМАНИЕ.** Для работы с несколькими задачами удобно упаковать объекты Callable в List<Future<T>>
  - Объект Future<T>
    - это объект потока, к нему прикрепляются задачи, используется для создания коллекций потоков
    - Future<?> объект потока для задач Runnable
    - Future<T> объект потока для задач Callable
  - Методы Future<>
    - get() приостановить основной поток, ждать результата до завершения потока клиента
    - isDone() проверка завершения потока клиента и готовности получения результата

- Пример. реализация интерфейса Callable<String>

```
public class TaskWithResult implements Callable<String> {
    private int id;
    public TaskWithResult(int id) {
        this.id = id;
    }
    @Override
    public String call() {
        for (int i = 0; i < 10; i++) {
            System.out.print("#"+id+"."+i+" ");
        }
        return "result of TaskWithResult "+id;
    }
}
```

[lesson\\_ch21/ch/ex05/codea](#)

- Применение. упаковка объектов Callable в List<Future<String>>

```
public class CallableDemo {
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool(); // создаем pool потоков
        ArrayList<Future<String>> results = new ArrayList<>();

        for (int i = 0; i < 10; i++) {
            results.add(exec.submit(new TaskWithResult(i))); // входной параметр =id
        } // загнали все потоки в список <Future<T>> и они сразу пошли на выполнение

        for (Future<String> fs : results) { // прогоняем список
            try {
                System.out.println(fs.get()); // получить результат с блокировкой
                if(fs.isDone()) {
                    System.out.println(fs.get()); // получить если готов, иначе далее
                }
            } catch (InterruptedException e) {
                System.out.println(e);
                return;
            } catch (ExecutionException e) {
                System.out.println(e);
            } finally {
                exec.shutdown(); // закрыть poll потоков по любому
            }
        }
    }
}
```

[lesson\\_ch21/ch/ex05/codea](#)

## Ожидание

- Ожидание
  - TimeUnit.MILLISECONDS.sleep() метод приостановки потока, может вызвать прерывание
  - вызывает interruptException в потоке клиента
  - прерывания в потоке надо отработать локально
- **ВНИМАНИЕ.** Так как Exceptions в потоках НЕ передаются в main() их надо отрабатывать ЛОКАЛЬНО
- Пример. реализация TimeUnit.sleep() в потоке выполнения

```
public class SleepingTask extends LiftOff {
    @Override
    public void run() {
        try {
            while (countDown-- > 0) {
                System.out.print(status()); // тоже самое
                TimeUnit.MILLISECONDS.sleep(100); // задержка в каждом потоке
            }
        } catch (InterruptedException e) {
            System.err.println("#" + id + " interrupted");
        }
    }
}
```

[lesson\\_ch21/ch/ex06/codea](#)

## Приоритет

- Приоритет
    - потоки с высоким приоритетом вызываются чаще, с низким реже
    - если приостановлено несколько потоков, первым будет запущен поток с высоким приоритетом
    - обычно потоки запускаются с приоритетом по умолчанию
  - **ВНИМАНИЕ.** Манипулирование приоритетом потоков ОШИБКА и не рекомендовано для использования
  - Пример. реализация уровней приоритетов
- ```
lesson_ch21/ch/ex07/codea
```
- ```
public class SimplePriorities implements Runnable {  
    private int countDown = 5;  
    private volatile double d; // без оптимизации  
    private int priority;  
  
    public SimplePriorities(int priority) {  
        this.priority = priority;  
    }  
    @Override  
    public void run() {  
        Thread.currentThread().setPriority(priority);  
        while (true) {  
            for (int i = 1; i < 100000; i++) {  
                d += (Math.PI + Math.E) / (double) i;  
                if (i % 1000 == 0) {  
                    Thread.yield(); // высокоприоритетный поток дал другим выполниться  
                }  
            }  
            System.out.println(this); // повторить 5 раз полный цикл  
            if (--countDown == 0) {  
                return;  
            }  
        }  
    }  
    public static void check() {  
        ExecutorService exec = Executors.newCachedThreadPool();  
        for (int i = 0; i < 5; i++) {  
            exec.execute(new SimplePriorities(Thread.MIN_PRIORITY)); // 5 процессов обычные  
        }  
        exec.execute(new SimplePriorities(Thread.MAX_PRIORITY)); // 1 процесс тяжелый  
        exec.shutdown();  
    }  
    @Override  
    public String toString() {  
        return Thread.currentThread() + ":" + countDown;  
    }  
}
```

## Уступки

- Уступки
    - Thread.yield() метод который рекомендует планировщику переключиться на другие процессы
  - **ВНИМАНИЕ.** Thread.yield() НЕ гарантированный сервис, поэтому НЕ РЕКОМЕНДУЕТСЯ к использованию.
  - Пример. реализация уступки высокоприоритетного процесса, каждый 1000 й цикл
  - ```
public void run() {  
    Thread.currentThread().setPriority(priority);  
    for (int i = 1; i < 100000; i++) {  
        if (i % 1000 == 0) {  
            Thread.yield(); // высокоприоритетный поток дал другим выполниться  
        }  
    }  
}
```
- ```
lesson_ch21/ch/ex07/codea
```

## Потоки демоны

- Потоки демоны
  - это потоки работающие в фоновом режиме, они никак не влияют на работу основной программы
  - программа ждет завершения обычных потоков
- Демоны свойства
  - программа игнорирует состояние потоков демонов при завершении
  - потоки демоны порождают демонов
- **ВНИМАНИЕ.** Блок finally{} ВЫПОЛНЯЕТСЯ и у Daemon и у не Daemon
- Пример. реализация потоков в фоновом режиме

[lesson\\_ch21/ch/ex07/codeb](#)

```
public class SimpleDaemons implements Runnable {  
    @Override  
    public void run() {  
        try {  
            while (true) { // ПОТОКИ демоны работают бесконечно.  
                TimeUnit.MILLISECONDS.sleep(100);  
                System.out.println(Thread.currentThread() + " >> " + this);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("sleep() interrupted");  
        }  
    }  
    public static void check() {  
        try {  
            for (int i = 0; i < 10; i++) {  
                Thread thread = new Thread(new SimpleDaemons());  
                thread.setDaemon(true); // переключить в режим демона  
                thread.start();  
            }  
            System.out.println("All daemons started");  
            TimeUnit.MILLISECONDS.sleep(100);  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

- Пример. реализация потока демона, порождающего другие потоки

[lesson\\_ch21/ch/ex07/coded](#)

```
public class DaemonSpawn implements Runnable {  
    @Override  
    public void run() {  
        while (true) { // просто бесконечный цикл демона  
            Thread.yield();  
        }  
    }  
}  
  
public class Daemon implements Runnable {  
    private Thread[] t = new Thread[10];  
    @Override  
    public void run() {  
        for (int i = 0; i < t.length; i++) {  
            t[i] = new Thread(new DaemonSpawn()); // создать поток с while(true){}  
            t[i].start();  
            System.out.println("DaemonSpawn "+i+" started, ");  
        }  
        for (int i = 0; i < t.length; i++) {  
            System.out.println("t["+i+"].isDaemon() : "+t[i].isDaemon()+" , ");  
        }  
        while (true) {  
            Thread.yield(); // тоже улетел в бесконечный цикл  
        }  
    }  
}
```

## Реализация многопоточного приложения

- Реализация многопоточного приложения, есть несколько способов
  - на базе класса Thread                      только для простых реализаций
  - на базе интерфейса Runnable              только для простых реализаций
  - на базе именованного внутреннего класса
  - на базе анонимного внутреннего класса
- **ВНИМАНИЕ.** В обоих случаях поток запускается в конструкторе, это опасно, другая задача может обратиться к объекту до завершения конструктора, поэтому ИСПОЛЬЗОВАТЬ Executor
- Пример. реализация приложения на базе наследования Thread [lesson\\_ch21/ch/ex10/codea](#)
- ```
public class SimpleThread extends Thread{
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {
        super(Integer.toString(++threadCount)); // задает имя потока
        start(); // запускает поток сразу же не ждет приглашения
    }
    @Override
    public void run() { // переопределяем метод run()
        while (true) { // бесконечный цикл
            System.out.print(this); // падаем в свою же toString "#"+..
            if (--countDown == 0) {
                return; // выпрыгиваем из цикла
            }
        }
    }
    public static void check() {
        for (int i = 0; i < 5; i++) {
            new SimpleThread(); // сразу и запуск и все остальное
        }
    }
    @Override
    public String toString() {
        return "#" + getName() + "(" + countDown + ") , "; // распечатка параметров потока
    }
}
```
- Пример. реализация приложения на базе интерфейса Runnable [lesson\\_ch21/ch/ex10/codea](#)
- ```
public class SelfManaged implements Runnable{
    private int countDown = 5;
    private Thread t = new Thread(this); // создает поток при создании задачи
    public SelfManaged() { // ВНИМАНИЕ запуск в конструкторе потенциально опасен
        t.start(); // запускает поток сразу же не ждет приглашения
    }
    @Override
    public void run() { // переопределяем метод run()
        while (true) { // бесконечный цикл
            System.out.print(this); // падаем в свою же toString "#"+..
            if (--countDown == 0) {
                return; // выпрыгиваем из цикла
            }
        }
    }
    public static void check() {
        for (int i = 0; i < 5; i++) {
            new SelfManaged(); // сразу и запуск и все остальное
        }
    }
    @Override
    public String toString() {
        return t.getName() + "(" + countDown + ") , "; // распечатка параметров потока
    }
}
```

## Lambda замена вызова внутреннего класса

- Lambda -> замена вызова внутреннего класса

- Пример. реализация Lambda «->» внутреннего класса внутри метода

[lesson\\_ch21/ch/ex10/exercise](#)

```
public static IGenerator<Integer> gen() { // реализация внутреннего класса
    return new IGenerator<Integer>() {
        @Override
        public Integer next() {
            return new Random().nextInt(5) + 5;
        }
    };
}
public static IGenerator<Integer> gen2() { //тоже самое но через lambda
    return () -> new Random().nextInt(5) + 5;
}
```

## Синхронизация методов

- Синхронизация методов

- о методы работающие между потоками объявляются synchronized

- Пример. реализация подобных методов

[lesson\\_ch21/ch/ex10/exercise](#)

```
public class ThreadMethod4 {
private static Random rnd = new Random();
private static ExecutorService exec;
public static synchronized void init() {
    if (exec == null) {
        exec = Executors.newCachedThreadPool();
    }
}
public static synchronized void shutdown() { // synchronized для работы между потоками
    if (exec != null) exec.shutdown();
    exec = null;
}
public static Future<Integer> runTask(int n) {
    return exec.submit(new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            int sum = 0;
            Fibonacci f = new Fibonacci();
            for (int i = 0; i < n; i++) {
                sum += f.next();
            }
            return sum;
        }
    });
}
public static void check() {
    ArrayList<Future<Integer>> list = new ArrayList<>();
    ThreadMethod4.init();
    for (int i = 0; i < 10; i++) {
        list.add(ThreadMethod4.runTask(rnd.nextInt(5) + 5)); // входной параметр n
    }
    int index = 0;
    for (Future<Integer> fs : list) { // прогоняем список
        try {
            System.out.print("#" + index++ + "." + fs.get() + " "); // получить с блокировкой
        } catch (Exception e) {
            System.out.println(e);
            return;
        } finally {
            ThreadMethod4.shutdown(); // закрыть poll потоков по любому
        }
    }
}
```

## Потоки и задачи в чем разница

- Потоки и задачи в чем разница
  - Thread — это класс который сам ничего не делает, он только управляет выполнением задачи
  - Runnable — интерфейс, который тоже сам ничего не делает
  - Callable — интерфейс, который сам ничего не делает
- Поток — это механизм управления задачей
- Задача это выполняемая программа, которая выполняет некую работу

## Присоединение к потоку

- Присоединение к потоку
  - join() — метод присоединения другого потока к данному, вызывает InterruptedException
  - если в потоке A вызывается B.join() => поток A ждет завершения потока B
- Пример. реализация присоединения одного потока к другому [lesson\\_ch21/ch/ex11/codea](#)
- ```
public class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper) {
        super(name); // Thread(name)
        this.sleeper = sleeper;
        start();
    }
    @Override
    public void run() {
        try {
            System.out.println(getName()+" join started");
            sleeper.join(); // присоединяем поток sleeper и ждем его завершения
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
        System.out.println(getName()+" join completed");
    }
}
```

## Чуткие пользовательские интерфейсы

- Чуткие пользовательские интерфейсы
  - то есть это интерфейсы которые быстро реагируют на пользователя
  - основной смысл, это перевести вычисления в отдельный поток
  - а в основном потоке заниматься вводом с клавиатуры
- Пример. реализация ввода в основном потоке, а вычислений в демоне [lesson\\_ch21/ch/ex11/codeb](#)

```
public class ResponseUI extends Thread {
    private volatile double d = 1;

    public ResponseUI() {
        setDaemon(true);
        start();
    }
    @Override
    public void run() {
        while (true) {
            d = d+(Math.PI + Math.E)/d;
        }
    }
    public void check() throws IOException {
        new ResponseUI();
        System.in.read();
        System.out.println(d);
    }
}
```

## Группы потоков

- Группы потоков ThreadGroup неудачный эксперимент и рекомендуется их не использовать
- ВНИМАНИЕ. Не использовать ThreadGroup группы потоков

## Перехват исключений

- Перехват исключений
  - перехват исключений из потока делается с помощью интерфейса Thread.UncaughtExceptionHandler
- Процедура создания перехватчика Exception из потока
  - создать класс перехватчика с интерфейсом Thread.UncaughtExceptionHandler
  - создать поток Thread
  - подключить перехватчик к потоку с помощью t.setUncaughtExceptionHandler()
  - как вариант задать единый перехватчик Thread.setDefaultUncaughtExceptionHandler()
- Пример. реализация перехватчика Exception

[lesson\\_ch21/ch/ex11/coded](#)

```
public class ExceptionThread2 implements Runnable{
    @Override
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("run() by "+t);
        System.out.println("eh = "+t.getUncaughtExceptionHandler()+" "+t.getName());
        throw new RuntimeException("Exception from Thread"+t.getName());
    }
}
public class MyUncaughtExceptionHandler implements Thread.UncaughtExceptionHandler {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("handler caught "+e);
    }
}
public class HandlerThreadFactory implements ThreadFactory {
    @Override
    public Thread newThread(Runnable r) {
        System.out.println("factory " + this + " creating new Thread");
        Thread t = new Thread(r);
        System.out.println("factory created "+t);
        t.setUncaughtExceptionHandler(new MyUncaughtExceptionHandler()); // set catcher
        System.out.println("factory eh = **"+t.getUncaughtExceptionHandler());
        return t;
    }
}
```

- Применение. Используется ThreadFactory для подключения обработчика

```
public static void check() {
    ExecutorService exec = Executors.newCachedThreadPool(new HandlerThreadFactory());
    exec.execute(new ExceptionThread2()); // с перехватчиком
    exec.shutdown(); // то есть нормально срабатывает и без перехвата
    Time.sleep(100);
}
```

- Пример. реализация обработчика по умолчанию для всех потоков

[lesson\\_ch21/ch/ex11/coded](#)

```
public static void check() { // обработчик по умолчанию
    Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandler());
    ExecutorService exec = Executors.newCachedThreadPool();
    for (int i = 0; i < 5; i++) {
        exec.execute(new ExceptionThread2());
    }
    exec.shutdown();
    Time.sleep(100);
}
```

## Совместное использование ресурсов

- Совместное использование ресурсов порождает проблемы
  - коллизии потоков при некорректном доступе к ресурсам
- Пример. реализация генератора Int с «инкремент» в качестве ресурса

[lesson\\_ch21/ch/ex11/codee](#)

- ```
public abstract class IntGenerator {  
    private volatile boolean canceled = false; // boolean атомарность volatile видимость  
    public abstract int next(); // абстрактный метод  
    public void cancel() { // флаг доступа к ресурсу  
        canceled = true;  
    }  
    public boolean isCanceled() {  
        return canceled;  
    }  
}  
public class EvenChecker implements Runnable {  
    private IntGenerator g;  
    private final int id;  
  
    public EvenChecker(IntGenerator g, int id) {  
        this.g = g;  
        this.id = id;  
    }  
    @Override  
    public void run() {  
        while (!g.isCanceled()) {  
            int val = g.next(); // генерит числа до первого нечетного  
            if (val % 2 != 0) {  
                System.out.println("#" + id + " " + val + " not even!");  
                g.cancel();  
            }  
        }  
    }  
    public static void test(IntGenerator gp, int count) {  
        System.out.println("Press Ctrl+C to exit");  
        ExecutorService exec = Executors.newCachedThreadPool();  
        for (int i = 0; i < count; i++) {  
            exec.execute(new EvenChecker(gp, i)); // id = i  
        }  
        Time.sleep(500);  
        exec.shutdown();  
    }  
}
```
- Применение. реализация IntGenerator

[lesson\\_ch21/ch/ex11/codee](#)

- ```
public class EventGenerator extends IntGenerator {  
    private volatile int currentEvenValue = 0;  
    @Override  
    public int next() {  
        ++currentEvenValue;  
        ++currentEvenValue;  
        return currentEvenValue;  
    }  
    public static void check() {  
        EvenChecker.test(new EventGenerator()); // генератор два инкремента, потоков 10  
    }  
}
```

## Разрешение спора за разделяемые ресурсы

- Разрешение спора за разделяемые ресурсы
  - mutex      блокировка ресурса, ресурс блокируется одним потоком
  - synchronized      метод встроенная поддержка блокировки , доступен одновременно одной задаче
  - **ВНИМАНИЕ.** private обязательно для field используемого в synchronized method
  - class      существует блокировка уровня класса
- Правило синхронизации
  - потоки которые работают с одним ресурсом должны быть синхронизированы
  - потоки работающие с одним ресурсом должны синхронизироваться по одной блокировке
- Пример. реализация синхронизации для IntGenerator [lesson\\_ch21/ch/ex11/codee](#)
- ```
public class SynchronizedEventGenerator extends IntGenerator {  
    private volatile int currentEvenValue = 0;  
    @Override  
    public synchronized int next() { // synchronized блокирует currentValue  
        ++currentEvenValue;  
        ++currentEvenValue;  
        return currentEvenValue;  
    }  
  
    public static void check() {  
        EvenChecker.test(new SynchronizedEventGenerator()); // генератор потоков 10  
    }  
}
```
- 

## Использование объектов Lock

- Использование объектов Lock
  - Lock это объект блокировки явного механизма mutex
- Преимущество перед synchronized в том, что
  - в блоке finally можно корректно освободить ресурсы
  - позволяет задавать цепную блокировку, когда следующий узел блокируется до снятия с текущего
- Методы
  - lock()      заблокировать участок метода
  - unlock()      снять блокировку, обязательно в finally{}, обязательно в блоке try{} команда return
  - trylock()      попытаться заблокировать в течение времени, если не получилось выйти
- **ВНИМАНИЕ.** Lock обязательно использовать с try\_finally и в секции try {} обязательно команда return
- Пример. реализация Lock объекта [lesson\\_ch21/ch/ex12/codea](#)

```
public class MutexEventGenerator extends IntGenerator{  
    private int currentEvenValue = 0;  
    private Lock lock = new ReentrantLock(); // объект Lock  
    @Override  
    public int next() {  
        lock.lock(); // заблокировать метод  
        try { // только для finally  
            ++currentEvenValue;  
            ++currentEvenValue;  
            return currentEvenValue;  
        }finally {  
            lock.unlock();  
        }  
    }  
    public static void check() {  
        EvenChecker.test(new MutexEventGenerator()); // генератор добавляет по 2, потоков 10  
    }  
}
```

- Пример. реализация работы с Lock по таймауту

[lesson\\_ch21/ch/ex12/codea](#)

```

    ○ объект создан в основном потоке
    ○ блокировку осуществляет Demon

• public class AttemptingLocking {
    private ReentrantLock lock = new ReentrantLock();
    public void untimed() {
        boolean captured = lock.tryLock();
        try {
            System.out.println("tryLock() :" + captured); // захвачен Lock или нет
        } finally {
            if (captured) {
                lock.unlock();
            }
        }
    }

    public void timed() {
        boolean captured = false;
        try {
            captured = lock.tryLock(2, TimeUnit.SECONDS); // попытка захвата 2ms

        } catch (InterruptedException e) {
            throw new RuntimeException();
        }
        try {
            System.out.println("tryLock(2,TimeUnit.SECONDS) :" + captured);

        } finally {
            if (captured) {
                lock.unlock();
            }
        }
    }

    public static void check() {
        final AttemptingLocking al = new AttemptingLocking();
        al.untimed(); // захватили блокировку
        al.timed();
        new Thread() {
            {
                setDaemon(true); // блок инициализации Thread
            }
            @Override
            public void run() {
                al.lock.lock(); // заблокировать код
                System.out.println("acquired");
                Time.sleep(100);
                al.lock.unlock();
                System.out.println("unlocked");
            }
        }.start();
        Thread.yield();
        Time.sleep(50);
        al.untimed();
        al.timed();

        Time.sleep(150);
        al.untimed();
        al.timed();
        //Time.sleep(150);
    }
}

```

- 
-

## Атомарность и Видимость изменений

- Атомарная операция
  - это операция, которую планировщик потоков не может прервать
  - рекомендуется блокировать любые операции и не полагаться на «атомарность»
  - возможна ситуация, когда атомарная операция кажется атомарной но не является такой на деле
- Атомарные операции
  - операции чтения и записи примитивных типов КРОМЕ long и double
- **ВНИМАНИЕ.** Атомарные операции потенциально опасны использовать НЕ РЕКОМЕНДУЕТСЯ
- Применять synchronized вместе с атомарными операциями
- Видимость изменений Volatile
  - volatile это обеспечение видимости изменений внесенных одним процессом, для других
  - означает, что сразу после записи все процессы «увидят» внесенные изменения
  - использовать всегда при обращении нескольких потоков, и хотя бы один из них запись по сути отменяет кэширование значений, поэтому изменения видимы сразу
- **ВНИМАНИЕ.** Volatile потенциально опасно использовать НЕ РЕКОМЕНДУЕТСЯ
- Применять synchronized вместе с volatile
- Volatile не работает
  - в пределах одной задачи нет смысла использовать volatile
  - если значение переменной зависит от предыдущего значения, volatile не работает
  - если значение одной переменной ограничено другой переменной

[lesson\\_ch21/ch/ex12/codea](#)

```
public class SerialNumberGenerator {  
    private static volatile int serialNumber = 0;  
    public synchronized static int nextSerialNumber() {  
        return serialNumber++;  
    }  
    // иначе без synchronized создает проблему  
    //    public static int nextSerialNumber2() {  
    //        return serialNumber++;  
    //    }  
    public class SerialNumberChecker {  
        private static final int SIZE = 10;  
        private static CircularTest serials = new CircularTest(1000); //массив 1000 номеров  
        private static ExecutorService exec = Executors.newCachedThreadPool();  
        static class SerialChecker implements Runnable {  
            @Override  
            public void run() {  
                while (true) {  
                    int serial = SerialNumberGenerator.nextSerialNumber();  
                    if (serials.conatins(serial)) {  
                        System.out.println("Duplicate: "+serial);  
                        break;  
                    }  
                    serials.add(serial);  
                }  
            }  
        }  
        public static void check() {  
            for (int i = 0; i < SIZE; i++) {  
                exec.execute(new SerialChecker());  
            }  
            Time.sleep(1000);  
            exec.shutdown();  
        }  
    }  
}
```

## Атомарные классы

- Атомарные классы – это классы атомарных переменных AtomicInteger, AtomicLong, AtomicReference
  - предназначены для атомарных операций вида compareAndSet(expectedValue, updateValue)
  - используются для машинной оптимизации кода

- Пример. реализация оптимизации с помощью атомарного класса

[lesson\\_ch21/ch/ex14/codea](#)

```
public class AtomicIntegerTest implements Runnable {
    private AtomicInteger i = new AtomicInteger(0);
    public int getValue() {
        return i.get();
    }
    private void evenIncrement() {
        i.addAndGet(2);
    }
    @Override
    public void run() {
        int count = 0;
        while (true) {
            evenIncrement();
            if (count++ > 1e8) {
                break;
            }
        }
        System.out.println("AtomicIntegerTest exit...");
    }
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicIntegerTest at = new AtomicIntegerTest();
        exec.execute(at);

        int count = 0;
        while (true) {
            int val = at.getValue();
            if (val % 2 != 0) {
                System.out.println(val);
                break;
            }
            if (count++ > 1e8) {
                break;
            }
        }
        exec.shutdown();
        Time.sleep(500);
    }
}
```

- Пример. реализации инкремента в виде Атомарного класса

[lesson\\_ch21/ch/ex14/codea](#)

```
public class AtomicEvenGenerator extends IntGenerator{
    private AtomicInteger currentEvenValue = new AtomicInteger(0);
    public int next() {
        return currentEvenValue.addAndGet(2);
    }

    public static void check() {
        EvenChecker.test(new AtomicEvenGenerator());
    }
}
```

## Критические секции

- Критические секции
  - критическая секция – это синхронизированная блокировка части кода метода
- Синтаксис
  - synchronized(syncObject) {} – секция к которой может иметь доступ только один поток
  - syncObject – объект блокировки, например this доступ к которому поток должен получить
- Пример. реализация критической секции на базе объекта this [lesson\\_ch21/ch/ex15/codea](#)

## Создание секций при помощи Lock

- Lock – объект, который используется для создания критической секции
  - **ВНИМАНИЕ.** объект Lock должен быть один для всех потоков которые его используют
  - объект Lock должен быть один для всех методов объекта которым нужна блокировка
  - **ВНИМАНИЕ.** Нельзя часть методов использовать synchronized, а часть Lock
  - если они завязаны в одном процессе
- Пример. реализация критической секции на базе объекта Lock [lesson\\_ch21/ch/ex15/codeb](#)

```
public class ExplicitPairManager1 extends PairManager {  
    private Lock lock = new ReentrantLock();  
    public Pair getPair() {  
        lock.lock();  
        try {  
            return new Pair(p.getX(), p.getY()); // создается копия  
        } finally {  
            lock.unlock();  
        }  
    }  
    public void increment() { // делается синхронным явно при помощи объекта Lock  
        lock.lock();  
        try {  
            p.incrementX();  
            p.incrementY();  
            store(getPair()); // инкрементируем и сохраняем копию в список  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

## Синхронизация критической секции по другим объектам

- Синхронизация по другим объектам [lesson\\_ch21/ch/ex15/codec](#)
  - обычно для критической секции используют текущий объект <this>
  - проблема в том, что при блокировании всего объекта доступ к одному методу блокирует все
  - при синхронизации по другим объектам возможен раздельный вход в объект синхронно
- Пример. реализация синхронизации по разным объектам [lesson\\_ch21/ch/ex15/codec](#)
- ```
public class ThreeSection2 extends ThreeSecBase {  
    Object syncObject = new Object();  
    Object syncObject2 = new Object();  
    public void m01() {  
        synchronized (syncObject) {  
            System.out.print("#1");  
        }  
    }  
    public void m02() {  
        synchronized (syncObject2) {  
            System.out.print("#2");  
        }  
    }  
}
```

## Локальная память потоков

- Локальная память потоков ThreadLocal
  - ThreadLocal – класс который отвечает за создание и управление локальной памяти потоков
  - устраниет конфликты за ресурсы созданием локальных ресурсов для потоков
  - например есть объект x и 5 потоков, создается 5 блоков памяти для x, по блоку на поток
- Пример. реализация локальной памяти потоков [lesson\\_ch21/ch/ex17/codea](#)
- ```
public class ThreadLocalVariableHolder { // одна статическая переменная на все
    private static ThreadLocal<Integer> value = new ThreadLocal<Integer>() {
        private Random rnd = new Random();
        @Override
        protected synchronized Integer initialValue() {
            return rnd.nextInt(1000);
        }
    };
    public static void increment() {           // функция несинхронная, это и не требуется
        value.set(value.get() + 1);
    }
    public static int get() {                 // функция несинхронная, это и не требуется
        return value.get();
    }
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        List<Future> list = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            list.add(exec.submit(new Accessor(i))); // отдаем id
        }
        try {
            TimeUnit.MILLISECONDS.sleep(1);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        for (Future future : list) {
            future.cancel(true);                // принудительно прерываем
        }
        exec.shutdown();
    }
}
```
- Организация доступа, потоки
- ```
public class Accessor implements Runnable{
    private final int id;
    public Accessor(int id) {
        this.id = id;
    }
    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            ThreadLocalVariableHolder.increment();
            System.out.println(this);
            Thread.yield();
        }
        System.out.println(this+" finished");
    }
    @Override
    public String toString() {
        return "#" + id + ":" + ThreadLocalVariableHolder.get();
    }
}
```

## Завершение задач

- Завершение задач
    - cancel() стандартный метод завершения
    - isCanceled() стандартный запрос статуса завершения
  - Ускоренные методы завершения задач
  - Пример. реализация быстрого завершения
    - ну на самом деле просто по флагу
- [lesson\\_ch21/ch/ex17/codeb](#)

## Завершение при блокировке

- Поток может находиться в одном из 4х состояний
  - Переходное new только во время создания
  - Активное runnable поток получил свой квант времени
  - Блокировки blocked поток заблокирован
  - Завершенное dead поток остановлен его задача завершена

## Блокированное состояние

- Поток приостановлен методом Time.sleep()
  - поток возобновит свою работу через заданное время
- Поток приостановлен методом wait()
  - поток будет стоять пока не получит уведомление notify(), notifyAll() или signal(), signalAll
- Поток ожидает завершения операции ввода, вывода
  - поток возобновит работу по завершении
- Поток пытается вызвать synchronized метод, но объект блокировки недоступен
  - поток возобновит работу после получения доступа
- **ВНИМАНИЕ.** Команды suspend(), resume(), stop() объявлены устаревшими

## Завершение задачи в Блокированном состоянии

- Завершение задачи в Блокированном состоянии
  - если поток заблокирован то выход из середины run() может потребовать освобождения ресурсов
  - для корректного прерывания потока используется метод Thread.interrupt()

## Прерывание

- Прерывание возбуждает исключение, чтобы корректно завершить заблокированный поток
  - Методы
    - interrupt() вызывает у потока InterrupException
    - interrupted() снимает состояние InterrupeException
  - interrupt()
    - Executor.shutdownNow() рекомендованный способ, для потоков execute(), submit()
    - используется для прерывания всех потоков запущенных службой
    - Future.cancel(true) рекомендованный способ, для потоков submit()
    - используется для остановки конкретного потока
    - Future<> объект создается службой при вызове метода submit()
  - Пример. реализация interrupt()
- [lesson\\_ch21/ch/ex18/codea](#)

## Снятие блокировки

- Снятие блокировки sleep()
  - отослать прерывание interrupt() потоку который ожидает таймаут
- Снятие блокировки ввода вывода
  - закрыть ресурс, который вызвал блокировку
  - использовать механизм разблокирования каналов <nio>
- **ВНИМАНИЕ.** при использовании NIO достаточно применить execute() и shutdownNow()
- Пример. реализация снятия блокировки закрытием ресурса [lesson\\_ch21/ch/ex18/codeb](#)
  - здесь для разблокирования потока закрывается ресурс, socket, который вызвал блокировку
- **public static void** check() {

```
try {  
    ExecutorService exec = Executors.newCachedThreadPool();  
    ServerSocket serverSocket = new ServerSocket(8080); // открыть socket  
    InputStream socketInput = new Socket("localhost", 8080).getInputStream();  
  
    exec.execute(new IOBlocked(socketInput)); // запустить поток с вводом из socket  
    TimeUnit.MILLISECONDS.sleep(100);  
    exec.shutdownNow(); // отослать прерывание  
    TimeUnit.SECONDS.sleep(1);  
    System.out.println("Closing " + socketInput.getClass().getSimpleName());  
    socketInput.close(); // закрываем поток ввода socketInput  
} catch (Exception e) { // автоматом разблокируется поток IOBlocked  
    throw new RuntimeException(e);  
}
```
- Пример. реализация разблокировки <nio> [lesson\\_ch21/ch/ex18/codec](#)
- **public class** NIOBlocked **implements** Runnable {

```
private final SocketChannel sc;  
public NIOBlocked(SocketChannel sc) {  
    this.sc = sc;  
}  
@Override  
public void run() {  
    try {  
        System.out.println("Waiting read() in" + this);  
        sc.read(ByteBuffer.allocate(100)); // операция чтения из SocketChannel  
    } catch (ClosedByInterruptException e) {  
        System.out.println("ClosedByInterruptException");  
    } catch (AsynchronousCloseException e) {  
        System.out.println("AsynchronousCloseException");  
    } catch (IOException e) {  
        System.out.println("IOException");  
    }  
}
```
- Применение.

```
public static void check() throws Exception {  
    try {  
        ExecutorService exec = Executors.newCachedThreadPool();  
        ServerSocket server = new ServerSocket(8082);  
        InetSocketAddress isa = new InetSocketAddress("localhost", 8082);  
        SocketChannel sc1 = SocketChannel.open(isa); // открыть два socket nio  
        SocketChannel sc2 = SocketChannel.open(isa);  
        Future<?> f = exec.submit(new NIOBlocked(sc1)); // запомнили, чтобы послать cancel  
        exec.execute(new NIOBlocked(sc2)); // запустили обычным способом  
        exec.shutdown(); // закрываем службу  
        TimeUnit.SECONDS.sleep(1); // прождали 1 секунду  
        f.cancel(true); // закрыли через персональное прерывание  
        TimeUnit.SECONDS.sleep(1); // прождали 1 секунду  
        sc2.close(); // закрыли через закрытие ресурса  
    }
```

## Способы остановить поток

- Способы остановить поток
  - нужно внутри потока создать поддержку interrupt(), а снаружи отправить запрос interrupt()
- main: InterruptGenerate
  - в основном методе main()
  - ExecutorService.submit() <> List<Future> Future.cancel()
  - List<Thread> Thread.interrupt()
  - ExecutorService.execute() <> ExecutorService.shutdownNow()
- run : InterruptSupport
  - в методе run() потока
  - TimeDelay <> try\_catch <InterruptException>
  - Thread.currentThread().isInterrupted() <> return
- Thread.interrupted() <> return равносильно Thread.currentThread().isInterrupted(true)

## Блокирование по mutex

- Блокирование по mutex
  - одна и та же задача может многократно захватывать mutex
- Пример. реализация MultiLock рекурсивного захвата методов одной задачей [lesson\\_ch21/ch/ex21/codea](#)
- Пример. реализация Interrupting2 прерывание блокировки ReentrantLock [lesson\\_ch21/ch/ex21/codea](#)

## Проверка прерывания

- Проверка прерывания
  - после выдачи потоку сигнала прерывания interrupt() он находится в этом состоянии «interrupted»
  - сброс состояния происходит либо через InterrupException либо через Thread.interrupted()
- Пример. реализация отработки прерывания потока [lesson\\_ch21/ch/ex21/codeb](#)
  - отработка блокированных участков кода или методов
  - отработка неблокированных участков кода или методов

```
public class Blocked3 implements Runnable {
    private volatile double d = 0.0;
    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                NeedsCleanup n1 = new NeedsCleanup(1);
                try {
                    System.out.println("Sleeping");
                    TimeUnit.SECONDS.sleep(1);
                    NeedsCleanup n2 = new NeedsCleanup(2);
                    try {
                        System.out.println("Calculating");
                        for (int i = 0; i < 2500000; i++)
                            d = d + (Math.PI + Math.E) / d;
                    } finally {
                        n2.cleanup();
                    }
                } finally {
                    n1.cleanup();
                }
            }
            System.out.println("Exiting via while() test");
        } catch (InterruptedException e) {
            System.out.println("Exiting via InterruptedException");
        }
    }
}
```

## Взаимодействие между задачами

[lesson\\_ch21/ch/ex21/codec](#)

- Взаимодействие между задачами handshaking
  - для использования совместно ресурса потоки используют mutex и синхронный доступ
  - для выполнения согласованной работы потоки используют handshaking через обмен сигналами
- Методы Object
- **ВНИМАНИЕ.** wait(), notify()
  - принадлежат к классу Object, можно использовать везде, не только в потомках Thread, Runnable
  - применять только в СИНХРОННОМ методе или блоке, иначе выдаст IllegalMonitorStateException
  - поток использующий wait() должен получить доступ к объекту блокировки до вызова wait()
- Описание
  - wait() приостановка потока с освобождением блокировки до сигналов notify(), notifyAll()
  - wait( time) приостановка потока на время с освобождением блокировки
  - выйти из wait(time) можно по сигналу notify(), notifyAll() или по истечении time
  - notify() сигнал одному из потоков который сидит в wait(), что можно продолжать работу
  - notifyAll() сигнал всем потокам которые сидят в wait(), что можно продолжать работу
  - **ВНИМАНИЕ.** Выбор потока по notify() произвольный
- Методы Condition
  - await()
  - signal()

## Отличие wait() от sleep()

- Отличие wait() от sleep()
  - Блокировка
  - Метод sleep() приостанавливает поток но НЕ СНИМАЕТ блокировку
  - Метод wait() приостанавливает поток и СНИМАЕТ блокировку на время ожидания
  - Выход
  - Метод sleep(time) выходит по времени или по прерыванию InterruptedException
  - Метод wait(time) выходит по времени или по сигналам notify() или notifyAll()

## Использование wait(), notifyAll()

- Метод wait() принадлежит к классу Object
  - принадлежат к классу Object, можно использовать везде, не только в потомках Thread, Runnable
  - применять только в СИНХРОННОМ методе или блоке, иначе выдаст IllegalMonitorStateException
  - поток использующий wait() должен получить доступ к объекту блокировки до вызова wait()
- Использование notifyAll()
  - для отправки сообщения notifyAll() объекту x его надо поместить в блок synchronized для x

```
	synchronized (s) {  
		s.notifyAll();  
}
```
- Пример. реализация взаимодействия потоков
- T2 сразу по <!waxOn> влетает в wait() , T1 делает операцию, set.waxOn, notifyAll() и по <waxOn> в wait()
- T2 выходит по notifyAll(), делает операцию, reset.waxOn, notifyAll() и по <!waxOn> влетает в wait()
- T1 выходит по notifyAll(), делает операцию, set.waxOn, notifyAll() и по <waxOn> влетает в wait()
- **ВНИМАНИЕ.** wait() заключен в цикл while() потому что ВАЖНО постоянно проверять условие ожидания
  - несколько задач, и другая задача может установить условие цикла и нужен повторный wait()
  - после notifyAll() другая задача может установить условие цикла и нужен повторный wait()
  - несколько задач, и задача должна убедиться что notifyAll() верный, иначе нужен повторный wait()

[lesson\\_ch21/ch/ex21/codec](#)

- Пример. реализация двух видов синхронизации по методу и по блоку

```

public class BallWait implements Runnable {
    private synchronized void checkWait() throws InterruptedException{
        wait();
    }
    public void run() {
        try {
            System.out.println("Ball runnned and waits");
            checkWait();
            System.out.println("Ball waits again");
            synchronized (this) {
                wait();
            }
        } catch (InterruptedException e) {
            System.out.println("Ball interrupted..");
        }
    }
}
public class MatchRun implements Runnable {
    BallWait ballWait;
    public MatchRun(BallWait ballWait) {
        this.ballWait = ballWait;
    }
    private void checkNotify() {
        synchronized (ballWait) {
            ballWait.notifyAll();
        }
    }
    public void run() {
        try {
            System.out.println("Match timer started");
            TimeUnit.MILLISECONDS.sleep(1500);
            System.out.println("Match issued notifyAll()");
            checkNotify();
            TimeUnit.MILLISECONDS.sleep(100);
            System.out.println("Match issued notifyAll() again");
            synchronized (ballWait) {
                ballWait.notifyAll();
            }
        } catch (InterruptedException e) {
            System.out.println("Match interrupted");
        }
    }
}

```

## Пропущенные сигналы

- Пропущенные сигналы
  - при синхронизации потоков возможна потеря сигналов
  - это ситуация когда сигнал notify() выдается раньше заточенного под него wait() в другом потоке
- Предотвращение гонки сигналов потоков T1 и T2
  - используется цикл while(condition) до входа в wait() в потоке T2
  - если T1 сработал раньше, то он изменит condition, и входа в wait() не будет, не пропустит while()
  - если T2 сработал раньше, то T2 входит в wait(), после чего T1 меняет условие и выдает notify()
- synchronized (sharedMonitor) {**
  - while (condition) {**
  - sharedMonitor.wait();**
- ВНИМАНИЕ.** ВСЕГДА заключать wait() в цикл while()
  - нет питания, получен сигнал notifyAll(), поток выходит из wait() и тут другой поток «сыел» питание
  - без while() текущий поток продолжит без питания, если есть while проверка перезапустить wait()

## Методы notify() и notifyAll()

- Методы notify() и notifyAll()
  - notify()      требует осторожности, запускает произвольную задачу из нескольких в ожидании
  - notifyAll()     активирует только те задачи, которые связаны на конкретный объект блокировки
- Пример. реализация проверки notifyAll()

[lesson\\_ch21/ch/ex23/codea](#)

```
public class Task implements Runnable {
    public static Blocker blocker = new Blocker();
    public void run() {
        blocker.waitingCall();
    }
}
public class Task2 implements Runnable {
    public static Blocker blocker = new Blocker();
    public void run() {
        blocker.waitingCall();
    }
}
public class Blocker {
    synchronized void waitingCall() {
        try {
            while (!Thread.interrupted()) { // бесконечный цикл ожидания по notify()
                wait(); // this объект блокировки
                System.out.println(Thread.currentThread()+" ");
            }
        } catch (InterruptedException e) {
        }
    }
    public synchronized void prod() {
        notify(); // по данному объекту
    }
    public synchronized void prodAll() {
        notifyAll(); // по данному объекту
    }
}
public class NotifyVsNotifyAll {
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i = 0; i < 5; i++) {
            exec.execute(new Task()); // 5 задач одного типа
        }
        exec.execute(new Task2()); // 1 задача другого типа
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            boolean prod = true;
            public void run() {
                if (prod) { // мигалка выдает каждые 400ms сигнал notify или notifyAll
                    System.out.print("\nnotify() ");
                    Task.blocker.prod();
                    prod = false;
                } else {
                    System.out.println("\nnotifyAll() ");
                    Task2.blocker.prodAll();
                    prod = true;
                }
            }
        }, 400, 400);
        Time.sleep(5000);
        timer.cancel();
        Time.sleep(500);
        Task2.blocker.prodAll();
        Time.sleep(500);
        exec.shutdownNow(); // отослать всем interrupt()
    }
}
```

## Синхронизация по объекту

- Синхронизация по объекту можно использовать любой объект
- Синхронизация делается в два этапа
  - первое это захватить в блоке synchronized(object) и послать object.wait()
  - второе это захватить в блоке synchronized(object) и послать object.notifyAll()
- Пример 1. использование объекта класса Ball ball = new Ball() самого в качестве объекта синхронизации
  - первое, захват synchronized(this) >> wait() то есть без ничего просто wait() означает this.wait()
  - второе , захват synchronized(ball) >> ball.notifyAll()
- Пример 2. использование внутри объекта Circle объекта класса Ball в качестве объекта синхронизации
  - первое, захват synchronized(ball) >> ball.wait() то есть без ничего просто wait() означает this.wait()
  - второе , захват synchronized(ball) >> ball.notifyAll()
- **ВНИМАНИЕ.** захват объекта означает, что и функции wait(), notifyAll() должны быть применены к нему
- **ВНИМАНИЕ.** если захват одного объекта в нескольких потоках, то да все потоки вскроются по notifyAll но все кроме избранных опять влетят в wait() по условиям

## Синхронизация по нескольким объектам одновременно НЕ РЕКОМЕНДУЕТСЯ

- Синхронизация по нескольким объектам одновременно НЕ РЕКОМЕНДУЕТСЯ
- Пример. реализация wait() синхронизации по нескольким объектам [lesson\\_ch21/ch/ex24/exercise](#)
- используются вложенные блоки synchronized
- **ВНИМАНИЕ.** НЕ РЕКОМЕНДУЕТСЯ но возможно

```
public class WaitPerson implements Runnable{
    private Restaurant2 restaurant;
    public WaitPerson(Restaurant2 restaurant) {
        this.restaurant = restaurant;
    }
    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) { // снимает флаг
                synchronized (this) { // waitPerson ждет
                    while (restaurant.mealPack.stat != MealStatus.READY) {
                        wait(); // this.wait ждет waitPerson.notifyAll
                    }
                }
                System.out.print("WaitPerson got "+restaurant.mealPack.meal+" ");
                synchronized (restaurant.busBoy) { // захватить busBoy
                    synchronized (restaurant.chef) { // захватить chef
                        restaurant.mealPack.stat = MealStatus.TOGARBAGE; //статус BusBoy
                        restaurant.chef.notifyAll(); // оповестили chef.wait()
                    }
                    restaurant.busBoy.notifyAll(); // оповестили busBoy.wait()
                }
            }
        } catch (InterruptedException e) {
            System.out.println("WaitPerson interrupted");
        }
    }
}
```

## Синхронизация по одному объекту

- Синхронизация по одному объекту
- Пример. реализация wait() синхронизации по нескольким объектам [lesson\\_ch21/ch/ex24/exercise](#)
  - используется расширенная блокировка по одному объекту и активное использование while()
- **ВНИМАНИЕ.** РЕКОМЕНДУЕТСЯ но возможен применению
- ```
public enum MealStatus {  
    EMPTY, READY, TOGARBAGE;  
  
}  
• public class Meal {  
    private final int orderNum;  
  
    public Meal(int orderNum) {  
        this.orderNum = orderNum;  
    }  
    @Override  
    public String toString() {  
        return "Meal " + orderNum;  
    }  
}  
• public class MealPack {  
    public Meal meal;  
    public MealStatus stat = MealStatus.EMPTY;  
  
}
```
- 
- Применение. синхронизация по одному объекту для всех [lesson\\_ch21/ch/ex24/exercise](#)
  - блокировка должна быть по этому объекту и в блоке wait() и в блоке notifyAll()
  - вызов команды wait() и команды notifyAll() должен быть по этому объекту
  - объект должен быть единым для всех участников
  - notifyAll() вскрывает всех участников, цикл while() вызывает заново wait() тем, кто должен ждать
- ```
public class WaitPerson implements Runnable{  
    private Restaurant restaurant;  
  
    public WaitPerson(Restaurant restaurant) {  
        this.restaurant = restaurant;  
    }  
    @Override  
    public void run() {  
        try {  
            while (!Thread.interrupted()) {  
                synchronized (restaurant.mealPack) {  
                    while (restaurant.mealPack.stat != MealStatus.READY) {  
                        restaurant.mealPack.wait();  
                    }  
                }  
                System.out.print("WaitPerson got "+restaurant.mealPack.meal+" ");  
                synchronized (restaurant.mealPack) {  
                    restaurant.mealPack.stat = MealStatus.TOGARBAGE; // сбросить meal  
                    restaurant.mealPack.notifyAll(); // вскрыть все mealPack.wait()  
                }  
            }  
        } catch (InterruptedException e) {  
            System.out.println("WaitPerson interrupted");  
        }  
    }  
}
```

## Производители и потребители

- Производители и потребители
  - создается единый класс координационный центр, который содержит все объекты
  - каждый из объектов использует сам себя как объект синхронизации
  - каждый из объектов использует смежные объекты как объекты синхронизации
  - так как все объединены через единый центр, имеют доступ друг к другу, могут послать сообщения
- Пример. реализация синхронизации

[lesson\\_ch21/ch/ex24/codea](#)

[lesson\\_ch21/ch/ex24/codea](#)

## Реализация асинхронной очереди

- Реализация асинхронной очереди
  - очередь создается отдельным объектом, доступ к ней по блокировке самой очереди
  - producer и consumer подключаются к очереди асинхронно
  - очередь использует wait() для защиты от переполнения или опустошения
- Пример. реализация синхронной очереди

[lesson\\_ch21/ch/ex24/codea](#)

```
public class FlowQueue<T>{  
    private Queue<T> queue = new LinkedList<T>();  
    private int maxSize;  
    public FlowQueue(int maxSize) {  
        this.maxSize = maxSize;  
    }  
    public synchronized void put(T v) throws InterruptedException {  
        while (queue.size() >= maxSize) {  
            System.out.println("maxsize wait");  
            wait();  
        }  
        queue.offer(v); // добавить элемент в очередь  
        notifyAll();  
    }  
    public synchronized T get() throws InterruptedException {  
        while (queue.isEmpty()) {  
            System.out.println("empty wait");  
            wait();  
        }  
        T returnVal = queue.poll(); // вытащить и удалить  
        notifyAll();  
        return returnVal;  
    }  
    public int getMaxSize() {  
        return maxSize;  
    }  
}
```

## Explicit Lock and Condition

- Explicit Lock and Condition
  - await() метод ожидания вызывается для приостановки выполнения задачи
  - signal или signalAll() метод оповещения что можно продолжить выполнение задачи
- Пример. реализация explicit Lock при помощи await(), signalAll()  
[lesson\\_ch21/ch/ex27/codea](#)
  - condition.signalAll выдергивает все потоки с lock.lock
  - именно while() с условием загоняет обратно те потоки, которые не должны проснуться

```
public class Car2 {  
    private Lock lock = new ReentrantLock();  
    private Condition condition = lock.newCondition();  
    private boolean waxOn = false;  
    public void waxed() {  
        lock.lock(); // замена synchronized  
        try {  
            waxOn = true;  
            condition.signalAll(); //замена notifyAll();  
        } finally {  
            lock.unlock();  
        }  
    }  
    public void buffed() {  
        lock.lock(); // замена synchronized  
        try {  
            waxOn = false;  
            condition.signalAll(); //замена notifyAll();  
        } finally {  
            lock.unlock();  
        }  
    }  
    public void waitForWaxing() throws InterruptedException {  
        lock.lock(); // замена synchronized  
        try {  
            while (waxOn == false) { // ПОСТОЯННО вызываем эту команду  
                condition.await(); // замена wait();  
            }  
        } finally {  
            lock.unlock();  
        }  
    }  
    public void waitForBuffing() throws InterruptedException {  
        lock.lock(); // замена synchronized  
        try {  
            while (waxOn == true) { // ПОСТОЯННО вызываем ОБЯЗАТЕЛЬНО  
                condition.await(); // wait();  
            }  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

## Производители и потребители и очереди

- Производители и потребители и очереди
  - организация взаимодействия через блокирующие очереди
  - задача клиента приостанавливается, если очередь пустая
  - задача клиента возобновляется, если в очереди появляются элементы

- Виды блокирующих очередей
  - LinkedBlockingQueue неограниченная по размеру очередь
  - ArrayBlockingQueue неограниченная по размеру очередь
  - BlockingQueue фиксированного размера очередь

- Пример. реализация взаимодействия через очередь BlockingQueue

[lesson\\_ch21/ch/ex28/exercise](#)

```
public class LiftOffFiller implements Runnable {  
    private BlockingQueue<LiftOff> rockets;  
    public LiftOffFiller(BlockingQueue<LiftOff> rockets) {  
        this.rocks  
    }  
    public void add(LiftOff liftOff) throws InterruptedException {  
        rockets.put(liftOff);  
    }  
    @Override  
    public void run() {  
        try {  
            for (int i = 0; i < 5; i++) {  
                add(new LiftOff());  
            }  
            TimeUnit.MILLISECONDS.sleep(100);  
        } catch (InterruptedException e) {  
            System.out.println("Waking from take()");  
        }  
        System.out.println("Exiting LiftOffFiller");  
    }  
}  
public class LiftOffRunner implements Runnable {  
    private BlockingQueue<LiftOff> rockets;  
    public LiftOffRunner(BlockingQueue<LiftOff> rockets) {  
        this.rocks  
    }  
    public void add(LiftOff liftOff) {  
        try {  
            rockets.put(liftOff);  
        } catch (InterruptedException e) {  
            System.out.println("interrupted");  
        }  
    }  
    @Override  
    public void run() {  
        try {  
            while (!Thread.interrupted()) {  
                LiftOff rocket = rockets.take(); // взять из очереди подвешивает если нет  
                rocket.run(); // запустить элемент в потоке  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Waking from take()");  
        }  
        System.out.println("Exiting LiftOffRunner");  
    }  
}  
• static void test(String msg, BlockingQueue<LiftOff> queue) {  
    ExecutorService exec = Executors.newCachedThreadPool();  
    exec.execute(new LiftOffRunner(queue));  
    exec.execute(new LiftOffFiller(queue));  
    exec.shutdownNow(); // остановить задачу  
}
```

## Очередь BlockingQueue с элементами toast

- Очередь BlockingQueue с элементами toast
- 
- Пример. реализация очереди на три операции
  - три операции подготовка тоста, намазывание масла, нанесение джема
  - для реализации объект тост передается между процессами BlockingQueue
- Пример. реализация обмена потоками своими данными между очередями
- ```
public class Toast {  
    public enum Status{DRY, BUTTERED, JAMMED}  
    private Status status = Status.DRY; // просто пустой хлеб  
    private final int id;  
    public Toast(int id) {  
        this.id = id;  
    }  
    public void butter() {  
        status = Status.BUTTERED;  
    }  
    public void jam() {  
        status = Status.JAMMED;  
    }  
    public Status getStatus() {  
        return status;  
    }  
    public int getId() {  
        return id;  
    }  
    @Override  
    public String toString() {  
        return "Toast " + id +": " + status;  
    }  
}
```
- Stage 1. реализация, создается очередь тостов, в очередь добавляется пустой тост, с идентификатором
- ```
public class Toaster implements Runnable {  
    private ToastQueue toastQueue; // очередь с объектами тост  
    private int count = 0;  
    private Random rnd = new Random();  
    public Toaster(ToastQueue toastQueue) { // внешний экземпляр очереди  
        this.toastQueue = toastQueue;  
    }  
    @Override  
    public void run() {  
        try {  
            while (!Thread.interrupted()) {  
                TimeUnit.MILLISECONDS.sleep(100+rnd.nextInt(500));  
                Toast t = new Toast(count++); // счетчик тостов  
                System.out.println(t); // распечатали  
                toastQueue.put(t); // положили в очередь  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Toaster interrupted");  
        }  
        System.out.println("Toaster off");  
    }  
}
```
- 
- 
- 
- 
- 

[lesson\\_ch21/ch/ex29/codea](#)

- Stage 2. реализация, тост берется из dryQueue, обрабатывается и сохраняется в очередь butteredQueue

```
public class Butterer implements Runnable {
    private ToastQueue dryQueue;           // очередь входящая с объектами тост
    private ToastQueue butteredQueue;      // очередь выходящая с объектами тост
    private int count = 0;
    private Random rnd = new Random();

    public Butterer(ToastQueue dryQueue, ToastQueue butteredQueue) {
        this.dryQueue = dryQueue;
        this.butteredQueue = butteredQueue;
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {

                Toast t = dryQueue.take(); // взять тост из очереди входящей
                t.butter(); // намазать масло
                System.out.println(t); // распечатали
                butteredQueue.put(t); // положить объект в очередь выходящую
            }
        } catch (InterruptedException e) {
            System.out.println("Butterer interrupted");
        }
        System.out.println("Butterer off");
    }
}
```

- Stage 3. реализация, тост из очереди butteredQueue, обрабатывается и кладется в очередь finishedQueue

```
public class Jammer implements Runnable {
    private ToastQueue butteredQueue;       // очередь входящая с объектами тост
    private ToastQueue finishedQueue;        // очередь выходящая с объектами тост
    private int count = 0;
    private Random rnd = new Random();

    public Jammer(ToastQueue butteredQueue, ToastQueue finishedQueue) {
        this.butteredQueue = butteredQueue;
        this.finishedQueue = finishedQueue;
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {

                Toast t = butteredQueue.take(); // взять тост из очереди входящей
                t.jam(); // нанести джем
                System.out.println(t); // распечатали
                finishedQueue.put(t); // положить объект в очередь выходящую
            }
        } catch (InterruptedException e) {
            System.out.println("Jammer interrupted");
        }
        System.out.println("Jammer off");
    }
}
```

- Stage 4. реализация, тест извлекается из finishedQueue, проверяется на ошибки и выводится результат

```

public class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    public Eater(ToastQueue finishedQueue) {
        this.finishedQueue = finishedQueue;
    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                Toast t = finishedQueue.take(); // взять готовый бутерброд из очереди
                if(t.getId() != counter++ || t.getStatus() != Toast.Status.JAMMED) {
                    System.out.println(">>> Error :" + t);
                    System.exit(1); // выдать закрытие всего конвейера
                } else {
                    System.out.println("Chomp! " + t);
                }
            }
        } catch (InterruptedException e) {
        }
        System.out.println("Eater off");
    }
}

```

- Применение. реализация основной программы

[lesson\\_ch21/ch/ex29/codea](#)

```

public class ToastOMATIC {
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        ToastQueue dryQueue = new ToastQueue();
        ToastQueue butteredQueue = new ToastQueue();
        ToastQueue finishedQueue = new ToastQueue();
        exec.execute(new Toaster(dryQueue));
        exec.execute(new Butterer(dryQueue, butteredQueue));
        exec.execute(new Jammer(butteredQueue, finishedQueue));
        exec.execute(new Eater(finishedQueue)); // вырубает головной поток
        Time.sleep(5000); // в случае ошибок Eater выходит
        exec.shutdownNow(); // сюда попадаем после выруба нормально
    }
}

```

- Пример. реализация обратного вызова

[lesson\\_ch21/ch/ex29/exercise](#)

```

public class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    Thread headThread;
    public Eater(ToastQueue finishedQueue, Thread headThread) {
        this.finishedQueue = finishedQueue;
        this.headThread = headThread;
    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                Toast t = finishedQueue.take(); // взять готовый бутерброд из очереди
                if(((t.getStatus() != Toast.Status.BUTTERED) &&
                   (t.getStatus() != Toast.Status.JAMMED))) {
                    System.out.println(">>> Error :" + t);
                    headThread.interrupt(); // выдать прерывание в головной поток
                } else {
                    System.out.println("Chomp! " + t);
                }
            }
        } catch (InterruptedException e) {
        }
        System.out.println("Eater off");
    }
}

```

## Использование каналов ввода вывода между потоками

- Каналы ввода вывода между потоками по сути это BlockingQueue
  - PipedWriter запись в канал
  - PipedReader чтение из канала
- Пример реализация pipes input output
- ```
public class Sender implements Runnable {  
    private Random rnd = new Random();  
    private PipedWriter out = new PipedWriter();  
    public PipedWriter getPipedWriter() {  
        return out;  
    }  
    @Override  
    public void run() {  
        try {  
            while (true) {  
                for(char c ='A';c< 'z';c++) { // выдать символы алфавита  
                    out.write(c);  
                }  
                TimeUnit.MILLISECONDS.sleep(500);  
            }  
        } catch (IOException e) {  
            System.out.println(e+" Sender write exception");  
        } catch (InterruptedException e) {  
            System.out.println(e+ " Sender sleep interrupted");  
        }  
    }  
}  
public class Receiver implements Runnable {  
    private Random rnd = new Random();  
    private PipedReader in;  
  
    public Receiver(Sender sender) {  
        try {  
            this.in = new PipedReader(sender.getPipedWriter());  
        } catch (IOException e) {  
            System.out.println(e+" Receiver exception ");  
        }  
    }  
    @Override  
    public void run() {  
        try {  
            while (true) {  
                System.out.println("Read: "+(char)in.read()+" , ");  
            }  
        } catch (IOException e) {  
            System.out.println(e + " Receiver read exception");  
        }  
    }  
}
```
- Применение.
- ```
public static void check() {  
    Sender sender = new Sender();  
    Receiver receiver = new Receiver(sender);  
    ExecutorService exec = Executors.newCachedThreadPool();  
    exec.execute(sender);  
    exec.execute(receiver);  
    try {  
        TimeUnit.MILLISECONDS.sleep(1000);  
    } catch (InterruptedException e) {  
        throw new RuntimeException(e);  
    }  
    exec.shutdownNow();  
}
```

## Взаимная блокировка

[lesson\\_ch21/ch/ex31/codea](#)

- Взаимная блокировка
  - если в программе больше одного потока, возможна ситуация закольцованной цепной блокировки
  - такая ситуация называется deadlock взаимной блокировкой
- Пример. реализация классической взаимной блокировки
  - когда каждый поток пытается сначала блокирует правый элемент, затем левый все блокируется
  - если хотя бы один блокирует левый, потом правый, круговой блокировки не наступает

```
private boolean taken = false;
public String name;
public Chopstick(String name) {
    this.name = name;
}
public synchronized void take(String s) throws InterruptedException{
    while (taken) { // если taken == false то проходит и ставит taken=true
        wait(); // если taken = true то lock and wait()
    } // тоже самое если просто wait() то lock по this
    taken = true;
}
public synchronized void drop() {
    taken = false; // сбросить
    notifyAll(); // lock идет по this вскрыть wait()
}
}

• public class Philosopher implements Runnable {
    private Chopstick left;
    private Chopstick right;
    private final int id;
    private final int pondFactor;
    private Random rnd = new Random();
    public Philosopher(Chopstick left, Chopstick right, int id, int pondFactor) {
        this.left = left;
        this.right = right;
        this.id = id;
        this.pondFactor = pondFactor;
    }
    private void pause() throws InterruptedException {
        if (pondFactor == 0) { // вылетает сразу паузы нет
            return;
        }
        TimeUnit.MILLISECONDS.sleep(rnd.nextInt(pondFactor * 250));
    }
    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                System.out.println(this + " " + "thinking");
                pause();
                right.take(id + " right"); // InterruptedException >> catched
                left.take(id + " left"); // InterruptedException >> catched
                System.out.println(this + " " + "eating");
                pause();
                right.drop();
                left.drop();
            }
        } catch (InterruptedException e) {
            System.out.println(this + " exiting via interrupt");
        }
    }
    @Override
    public String toString() {
        return "Philosopher " + id;
    }
}
```

- Применение.

[lesson\\_ch21/ch/ex31/codea](#)

```

public static void check() {
    int ponder = 0;
    int size = 5;
    ExecutorService exec = Executors.newCachedThreadPool();
    Chopstick[] sticks = new Chopstick[size];
    for (int i = 0; i < sticks.length ; i++) {
        sticks[i] = new Chopstick("chop_" + i);
    }
    for (int i = 0; i < size ; i++) {
        exec.execute(new Philosopher(sticks[i],sticks[(i+1)%size],i,ponder));
    }
    try {
        TimeUnit.MILLISECONDS.sleep(5000); // 5 сек
        // System.in.read();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
    exec.shutdownNow();
}

```

## Взаимная блокировка условия

- Существует 4 условия взаимной блокировки
  - взаимное исключение. по крайне мере один ресурс должен быть в дефиците
  - по крайней мере один поток должен ожидать ресурса удерживаемого другим процессом
  - ресурс нельзя насилино отбирать, все потоки должны освобождать ресурсы естественным путем
  - должно быть в наличии круговое ожидание
- Способы борьбы с взаимной блокировкой
  - нарушить любое из 4х условий и взаимной блокировки не возникнет
- Пример. в задаче с палочками есть два способа решения для нарушения условия кругового ожидания
  - первое, один из профессоров берет сначала левую палочку затем правую, все остальные наоборот
  - второе, любой профессор берет первой палочку с наименьшим номером
- Способ 1. реализация DLDPhilosophers2 в методе run() объекта Philosopher [lesson\\_ch21/ch/ex31/codeb](#)
  - Philosopher\_0 берет палочки иначе, остальные одинаково, этого достаточно
- ```

if (id == 0) {
    left.take(id + " left");           // InterruptedException >> catched
    right.take(id + " right");         // InterruptedException >> catched
} else {
    right.take(id + " right");         // InterruptedException >> catched
    left.take(id + " left");           // InterruptedException >> catched
}

```
- Способ 2. реализация DLDPhilosophers3 в головном методе check() [lesson\\_ch21/ch/ex31/codeb](#)
  - Philosoper\_0 .. Philospher\_3 берут первой палочку 0, а другую по порядку
  - Philosopher\_4 должен брать 0,0 что заблокирует всех, поэтому он повторяет любого из Philosophers
- ```

int [][] idx = new int[][] { {0,1},{0,2},{0,3},{0,4},{0,1} };
for (int i = 0; i < size ; i++) {
    exec.execute(new Philosopher(sticks[idx[i][0]],sticks[idx[i][1]],i,ponder));
}

```
- 
-

## Классы для параллельных вычислений

### CountDownLatch Class

- CountDownLatch служит для ожидания одной группой задач завершения другой группы задач
  - CountDownLatch объект получает значение счетчика,
  - CountDownLatch.await() блокирует задачу пока счетчик больше нуля
  - CountDownLatch.countDown() уменьшает счетчик, объект одноразовый, счетчик нельзя сбросить
- Пример. реализация CountDownLatch [lesson\\_ch21/ch/ex32/codea](#)

```
public class TaskPortion implements Runnable { // задача меняет счетчик CountDownLatch
    private static int counter = 0;
    private final int id = counter++;
    private static Random rnd = new Random();
    private final CountDownLatch latch;
    public TaskPortion(CountDownLatch latch) {
        this.latch = latch;
    }
    public void doWork() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(rnd.nextInt(1000) + 10);
        System.out.println(this + " completed");
    }
    public void run() {
        try {
            doWork();
            latch.countDown();
        } catch (InterruptedException e) {
            System.out.println(this+"interrupted");
        }
    }
    public String toString() {
        return String.format("%-3d ", id); // первый аргумент
    }
}
public class WaitingTask implements Runnable { // задача в ожидании пока счетчик > 0
    private static int counter = 0;
    private final int id = counter++;
    private final CountDownLatch latch;
    public WaitingTask(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            System.out.println("Latch barrier passed for "+this);
        } catch (InterruptedException e) {
            System.out.println(this+"interrupted");
        }
    }
    public String toString() {
        return String.format("Waiting task %-3d ", id); // первый аргумент
    }
}
public class CountDownLatchDemo {
    private static final int SIZE = 10;
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        CountDownLatch latch = new CountDownLatch(SIZE); // 10 тиков
        for (int i = 0; i < SIZE; i++)
            exec.execute(new WaitingTask(latch)); // все 10 задач ждут счетчика
        for (int i = 0; i < SIZE; i++)
            exec.execute(new TaskPortion(latch));
        System.out.println("Launched all tasks");
        exec.shutdown();
    }
}
```

- Пример. реализация синхронизации потоков при помощи CountDownLatch [lesson\\_ch21/ch/ex32/exercise](#)
- **ВНИМАНИЕ.** отличный пример

## Потоковая безопасность и библиотеки

- **ВНИМАНИЕ.** Java НЕ ДАЕТ информации о методах которые безопасно применять в потоках

### CyclicBarrier Class

- CyclicBarrier служит для организации одновременного завершения группы задач
  - объект класса CyclicBarrier многоразовый, его счетчик можно сбросить
- Принцип работы CyclicBarrier
  - при создании задается число потоков и объект Runnable()
  - каждый из потоков после выполнения действий останавливается в await()
  - когда число потоков остановленных await() сравняется с счетчиком CyclicBarrier запустить Runnable
  -
- Пример. реализация работы CyclicBarrier [lesson\\_ch21/ch/ex33/codea](#)

```

public class Horse implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private int strides = 0;
    private static Random rnd = new Random();
    private CyclicBarrier barrier;
    public Horse(CyclicBarrier barrier) {
        this.barrier = barrier;
    }
    public synchronized int getStrides() {
        return strides;
    }
    public String tracks() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < getStrides(); i++) {
            sb.append("*");
        }
        sb.append(id);
        return sb.toString();
    }
    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                synchronized (this) {
                    strides += rnd.nextInt(3); // 0,1,2 изменение и запрос
                    СИНХРОНИЗИРОВАНЫ
                }
                barrier.await();
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch (InterruptedException e) {
            System.out.println(this + " interrupted");
        } catch (BrokenBarrierException e) {
            throw new RuntimeException(e);
        }
    }
    @Override
    public String toString() {
        return "Horse " + id + " ";
    }
}

```

-

- Применение.
  - CyclicBarrier.run() автоматом перезапускается, пока внешние задачи await() выбирают весь счетчик
  - чтобы остановить CyclicBarier надо остановить все внешние задачи и выйти из CyclicBarrier.run()
- ```
public class HorseRace {
    static final int FINISH_LINE = 15;
    private List<Horse> horses = new ArrayList<>();

    private ExecutorService exec = Executors.newCachedThreadPool();
    private CyclicBarrier barrier;

    private class HorseRun implements Runnable {
        private final int pause;

        public HorseRun(int pause) {
            this.pause = pause;
        }

        @Override
        public void run() {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < FINISH_LINE; i++) {
                sb.append("=");
            }
            System.out.println(sb);
            // print horse tracks
            for (Horse horse : horses) {
                System.out.println(horse.tracks());
            }
            // check horse run
            for (Horse horse : horses) {
                if (horse.getStrides() >= FINISH_LINE) {
                    System.out.println(horse + "won!");
                    exec.shutdownNow(); // отрубить все потоки
                    return;
                }
            }
            try {
                TimeUnit.MILLISECONDS.sleep(pause);
            } catch (InterruptedException e) {
                System.out.println("barrier-action sleep interrupted");
            }
        }
    }

    public HorseRace(int nHorses, int pause) {
        barrier = new CyclicBarrier(nHorses, new HorseRun(pause));
        for (int i = 0; i < nHorses; i++) {
            Horse horse = new Horse(barrier);
            horses.add(horse); // добавить в список класса
            exec.execute(horse);
        }
    }

    public static void check() {
        int nHorses = 7;
        int pause = 100;
        new HorseRace(nHorses, pause); // запуск гонок
    }
}
```
- 
-

## DelayQueue Class

- DelayQueue это объект неограниченной BlockingQueue которая реализует интерфейс Delayed
  - объект может быть извлечен из очереди только после определенной задержки
  - если не вышло время задержки у нескольких объектов, то poll() вернет null
- Пример. реализация работы с DelayQueue [lesson\\_ch21/ch/ex33/codeb](#)
  - принцип работы, в очередь DelayQueue закладываются клиенты Runnable
  - далее они извлекаются методом take() по одному и отдаются на выполнение
  - время задержки и сортировку очередь делает самостоятельно
  - чтобы все работало требуется переопределить методы интерфейса Delayed
- ```
public class DelayedTask implements Runnable, Delayed {  
    private static int counter = 0;  
    private final int id = counter++;  
    private final int delta;  
    private final long trigger;  
    protected static List<DelayedTask> sequence = new ArrayList<>();  
  
    public DelayedTask(int delta) { // при организации добавляет себя в очередь static  
        this.delta = delta;  
        this.trigger = System.nanoTime() +  
            TimeUnit.NANOSECONDS.convert(delta, TimeUnit.MILLISECONDS);  
        sequence.add(this);  
    }  
  
    @Override  
    public long getDelay(TimeUnit unit) {  
        return unit.convert(trigger - System.nanoTime(), TimeUnit.NANOSECONDS); // в ns  
    }  
  
    @Override  
    public int compareTo(Delayed o) {  
        DelayedTask that = (DelayedTask) o;  
        if (trigger < that.trigger) {  
            return -1;  
        }  
        if (trigger > that.trigger) {  
            return 1;  
        }  
  
        return 0;  
    }  
  
    @Override  
    public void run() {  
        System.out.print(this + " ");  
    }  
  
    public String summary() {  
        return "(" + id + ":" + delta + ")";  
    }  
  
    @Override  
    public String toString() {  
        return String.format("[%4d] ", delta) + "Task " + id;  
    }  
}
```

- EndSentinel такой же, как остальные но с другим функционалом [lesson\\_ch21/ch/ex33/codeb](#)
- ```

public class EndSentinel extends DelayedTask {
    private ExecutorService exec;

    public EndSentinel(int delta, ExecutorService exec) {
        super(delta);
        this.exec = exec;
    }

    @Override
    public void run() {
        System.out.println("\nStarted EndSentinel");
        for (DelayedTask delayedTask : sequence) {
            System.out.print(delayedTask.summary() + " ");
        }
        System.out.println();
        System.out.println(this + " calling shutdownNow()");
        exec.shutdownNow();
    }
}

```
- DelayedTaskConsumer класс выборки и запуска из очереди DelayQueue [lesson\\_ch21/ch/ex33/codeb](#)
- ```

public class DelayedTaskConsumer implements Runnable {
    private DelayQueue<DelayedTask> queue;

    public DelayedTaskConsumer(DelayQueue<DelayedTask> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                queue.take().run(); // очередь не отдаст объект пока не вышло время задержки
            }
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
        System.out.println("Finished DelayedTaskConsumer");
    }
}

```
- Применение. [lesson\\_ch21/ch/ex33/codeb](#)
- ```

public class DelayQueueDemo {
    public static void check() {
        Random rnd = new Random();
        ExecutorService exec = Executors.newCachedThreadPool();
        DelayQueue<DelayedTask> queue = new DelayQueue<>();
        for (int i = 0; i < 10; i++) {
            queue.put(new DelayedTask(rnd.nextInt(5000)+100)); // от 10ms до 1000ms
        }
        exec.execute(new DelayedTaskConsumer(queue)); // запустить все задачи на
        отработку
        queue.add(new EndSentinel(5100, exec)); // точка останова
    }
}

```
- - 
  - 
  - 
  - 
  -

## PriorityBlockingQueue Class

- PriorityBlockingQueue Class
  - это приоритетная блокирующая очередь, элементы извлекаются в порядке приоритета
  -
- Пример. реализация PriorityBlockingQueue
  - PrioritizedTask исполняемая задача с заданным приоритетом
  - EndSentinel выполняется с самым низким <-1> приоритетом, последний

[lesson\\_ch21/ch/ex33/codec](#)

[lesson\\_ch21/ch/ex33/codec](#)

```
public class PrioritizedTask implements Runnable, Comparable<PrioritizedTask> {  
    private static int counter = 0;  
    private final int id = counter++;  
    private Random rnd = new Random();  
    private final int priority;  
    protected static List<PrioritizedTask> sequence = new ArrayList<>();  
  
    public PrioritizedTask(int priority) {  
        this.priority = priority;  
        sequence.add(this);  
    }  
    @Override  
    public int compareTo(PrioritizedTask o) {  
        return (priority < o.priority) ? 1 :  
            (priority > o.priority) ? -1 : 0; // сокращенная форма  
    }  
    @Override  
    public void run() {  
        try {  
            TimeUnit.MILLISECONDS.sleep(250);  
        } catch (InterruptedException e) {  
            System.out.println(this + " interrupted");  
        }  
        System.out.println(this);  
    }  
    public String summary() {  
        return "(" + id + ":" + priority + ")";  
    }  
    public static class EndSentinel extends PrioritizedTask {  
        private ExecutorService exec;  
  
        public EndSentinel(ExecutorService exec) {  
            super(-1); // самый низкий приоритет, вызывать последним  
            this.exec = exec;  
        }  
        @Override  
        public void run() {  
            int count = 0;  
            for (PrioritizedTask pTask : sequence) {  
                System.out.print(pTask.summary() + " ");  
                if (++count % 5 == 0) {  
                    System.out.println();  
                }  
            }  
            System.out.println();  
            System.out.println(this + " calling shutdownNow()");  
            exec.shutdownNow();  
        }  
    }  
    @Override  
    public String toString() {  
        return String.format("[%3d]", priority) +  
            "Task " + id;  
    }  
}
```

- PrioritizedTaskProducer заполняет очередь задач

[lesson\\_ch21/ch/ex33/codec](#)

- причем заполнение идет параллельно с выборкой задач

```
public class PrioritizedTaskProducer implements Runnable {
    private Queue<Runnable> queue;
    private Random rnd = new Random();
    private ExecutorService exec;
    public PrioritizedTaskProducer(Queue<Runnable> queue,
                                    ExecutorService exec) {
        this.queue = queue;
        this.exec = exec;
    }
    @Override
    public void run() {
        try { // очередь неограниченная, блокировки никогда не происходит
            for (int i = 0; i < 20; i++) { // случайные приоритеты
                queue.add(new PrioritizedTask(rnd.nextInt(10))); // приоритет от 0 до 9
                Thread.yield();
            }
            for (int i = 0; i < 10; i++) { // высокие приоритеты
                TimeUnit.MILLISECONDS.sleep(250);
                queue.add(new PrioritizedTask(10)); // приоритет от 10
            }
            for (int i = 0; i < 10; i++) { // по нарастающему приоритету
                queue.add(new PrioritizedTask(i)); // приоритет от 0 до 9
            }
            queue.add(new PrioritizedTask.EndSentinel(exec)); // приоритет -1 последний
        } catch (InterruptedException e) {
            System.out.println("PrioritizedTaskProducer interrupted");
        }
        System.out.println("Finished PrioritizedTaskProducer");
    }
}
```

- PrioritizedTaskConsumer запускает задачи из очереди, параллельно с заполнением очереди продюсером

```
public class PrioritizedTaskConsumer implements Runnable {
    private PriorityBlockingQueue<Runnable> queue;
    public PrioritizedTaskConsumer(PriorityBlockingQueue<Runnable> queue) {
        this.queue = queue;
    }
    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                queue.take().run(); //
            }
        } catch (InterruptedException e) {
            System.out.println("PrioritizedTaskConsumer interrupted");
        }
        System.out.println("Finished PrioritizedTaskConsumer");
    }
}
```

- Применение.

[lesson\\_ch21/ch/ex33/codec](#)

- ```
public class PriorityBlockingQueueDemo {
    public static void check() {
        Random rnd = new Random();
        ExecutorService exec = Executors.newCachedThreadPool();
        PriorityBlockingQueue<Runnable> queue = new PriorityBlockingQueue<>();
        exec.execute(new PrioritizedTaskProducer(queue, exec)); // заполнить очередь
        exec.execute(new PrioritizedTaskConsumer(queue)); // отработать очередь
    }
}
```

## Управление GreenHouse на базе SchedulerExecuter

- Управление GreenHouse на базе SchedulerExecuter
  - общий смысл в складывании объектов Runnable в ScheduleExecutor
  - запуск задачи однократно ScheduleExecutor.schedule()
  - запуск задачи с повторами ScheduleExecutor.scheduleAtFixedRate()
  - останов планировщика ScheduleExecutor.shutdownNow()
- Пример. реализация управления на базе SchedulerExecutor [lesson\\_ch21/ch/ex33/coded](#)
- **ВНИМАНИЕ.** мощный пример реализации планировщика
- Пример. реализация управления на базе DelayQueue [lesson\\_ch21/ch/ex33/exercise](#)
  - **ВНИМАНИЕ.** вариант официального решения отличается но использует честное наследование
- **ВНИМАНИЕ.** мощный пример реализации планировщика на базе очереди

## Semaphore

- Semaphore      объект блокировки со счетчиком
  - позволяет обращаться к ресурсу n задачам одновременно
- Пример. реализация доступа к ресурсу через объект Semaphore [lesson\\_ch21/ch/ex34/codea](#)
- CheckoutTask поток использующий объект Semaphore

```
public class CheckoutTask<T> implements Runnable {  
    private static int counter = 0;  
    private final int id = counter++;  
    private Pool<T> pool;  
    public CheckoutTask(Pool<T> pool) {  
        this.pool = pool;  
    }  
    public void run() {  
        try {  
            T item = pool.checkOut(); // заняли  
            System.out.println(this + " checked out " + item);  
            TimeUnit.MILLISECONDS.sleep(1000);  
            System.out.println(this + " checked in " + item);  
            pool.checkIn(item); // освободили  
  
        } catch (InterruptedException e) {  
            System.out.println("interrupted");  
        }  
        System.out.println(this + " finished");  
    }  
    public String toString() {  
        return "CheckoutTask " + id;  
    }  
}
```

- Fat класс объекта с тяжелым конструктором [lesson\\_ch21/ch/ex34/codea](#)

```
public class Fat {  
    private volatile double d;  
    private static int counter = 0;  
    private final int id = counter++;  
    public Fat() { // типа тяжелый конструктор объекта  
        for (int i = 1; i < 10000 ; i++) {  
            d += (Math.PI + Math.E) / (double)i;  
        }  
    }  
    public void operation() {  
        System.out.println(this);  
    }  
    public String toString() {  
        return "Fat " + id;  
    }  
}
```

- Pool класс ресурса с объектом Semaphore

[lesson\\_ch21/ch/ex34/codea](#)

- Semaphore отрабатывает захват ресурса и раз задачами
- выделение ресурса каждой задачи сделано вручную через boolean[]

```

• public class Pool<T> {
    private int size;
    private List<T> items = new ArrayList<T>();
    private volatile boolean[] checkedOut;
    private Semaphore available;

    public Pool(Class<T> classObject, int size) {
        this.size = size;
        checkedOut = new boolean[size];
        available = new Semaphore(size, true);
        for (int i = 0; i < size; i++) {
            try {
                items.add(classObject.newInstance()); // Reflections
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }

    public synchronized T getItem() {
        for (int i = 0; i < size; i++) {
            if (!checkedOut[i]) {
                checkedOut[i] = true; // в общем сами контролируем список
                return items.get(i);
            }
        }
        return null; // объекта не нашли, все заняты
    }

    public synchronized boolean releaseItem(T item) {
        int index = items.indexOf(item); // поискать объект в списке
        if (index == -1) {
            return false; // не нашли объект и не освободили
        }
        if (checkedOut[index]) {
            checkedOut[index] = false;
            return true;
        }
        return false; // объект не выдавали, поэтому не освободили
    }

    public void checkIn(T item) {
        if (releaseItem(item)) {
            available.release(); // освободить семафор, сдать объект
        }
    }

    public T checkOut() throws InterruptedException { // занять семафор
        available.acquire(); // если объекта нет, то здесь блокировка
        return getItem();
    }
}

```

- Применение

[lesson\\_ch21/ch/ex34/codea](#)

- полное тестирование объекта Semaphore
- освобождение ресурса отрабатывает программа пользователя
- Semaphore.acquire() блокирует работу потока
- Semaphore.release() поток не блокирует

```

• public class SemaphoreDemo {
    private static final int SIZE = 25;

    public static void check() {
        try {
            final Pool<Fat> pool = new Pool<>(Fat.class, SIZE);
            ExecutorService exec = Executors.newCachedThreadPool();
            for (int i = 0; i < SIZE; i++) {
                exec.execute(new CheckoutTask<>(pool));
            }
            System.out.println("All CheckoutTasks created");
            List<Fat> list = new ArrayList<>();
            // main checks out all entries
            for (int i = 0; i < SIZE; i++) {
                Fat f = pool.checkOut();
                System.out.println(i + ": main() thread checked out " + f); // распечатать объект
            }
            Future<?> blocked = exec.submit(new Runnable() {
                @Override
                public void run() { // так как все объекты выданы, то Thread блокируется
                    try {
                        pool.checkOut();
                    } catch (InterruptedException e) {
                        System.out.println("Blocked checkOut() interrupted");
                    }
                    System.out.println("Blocked thread finished");
                }
            });
            TimeUnit.MILLISECONDS.sleep(1000);

            blocked.cancel(true); // выдрать из блокировки
            TimeUnit.MILLISECONDS.sleep(100);
            System.out.println("Checking in objects in list:" + list);
            for (Fat fat : list) {
                pool.checkIn(fat);
            }
            for (Fat fat : list) { // повторное впихивание не блокирует
                pool.checkIn(fat);
            }

            exec.shutdown(); // прекратить работу
            System.out.println("Shutdown all");
            TimeUnit.MILLISECONDS.sleep(100);

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}

```

## Exchanger

- Exchanger барьер, который меняет местами объекты двух задач
  - при входе в барьер задача использует один объект
  - при выходе из барьера задача использует другой объект

- Пример. реализация Exchanger

[lesson\\_ch21/ch/ex34/codeb](#)

```
public class ExchangerProducer<T> implements Runnable {
    private IGenerator<T> generator;
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    public ExchangerProducer(Exchanger<List<T>> ex, IGenerator<T> gen, List<T> holder) {
        this.exchanger = ex;
        this.generator = gen;
        this.holder = holder;
    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                for (int i = 0; i < ExchangerDemo.size; i++) {
                    holder.add(generator.next()); // заполняется локальный список
                }
                System.out.println("producer made:" + holder);
                holder = exchanger.exchange(holder); // полный список меняется с пустым
            }
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
    }
}
public class ExchangerConsumer<T> implements Runnable {
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    private volatile T value;
    public ExchangerConsumer(Exchanger<List<T>> exchanger, List<T> holder) {
        this.exchanger = exchanger;
        this.holder = holder;
    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                holder = exchanger.exchange(holder); // получить список от producer
                System.out.println("consumer got :" + holder);
                for (T t : holder) {
                    value = t;
                    holder.remove(t); // удаляем, так как потом отдадим producer
                }
            }
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
    }
}
public class ExchangerDemo {
    public static int size = 10;
    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        Exchanger<List<Fat>> xc = new Exchanger<>();
        List<Fat> producerList = new CopyOnWriteArrayList<>();
        List<Fat> consumerList = new CopyOnWriteArrayList<>();
        exec.execute(new ExchangerProducer<>(xc, new Fat.FatFactory(), producerList));
        exec.execute(new ExchangerConsumer<>(xc, consumerList));
        Time.sleep(1);
        exec.shutdownNow();
    }
}
```

## Моделирование

- Моделирование параллельное
    - при параллельном моделировании каждый компонент представлен отдельной задачей
  - Модель кассира
    - классическая модель, число объектов и время обслуживания случайно, число серверов ограничено
  - Пример. реализация модели кассира в банке
  - ```
public class Customer {  
    private final int serviceTime;  
    public Customer(int serviceTime) {  
        this.serviceTime = serviceTime;  
    }  
    public int getServiceTime() {  
        return serviceTime;  
    }  
    public String toString() {  
        return "[" + serviceTime + "]";  
    }  
}
```
  - Очередь клиентов на базе ArrayBlockingQueue() фиксированного размера
  - ```
public class CustomerLine extends ArrayBlockingQueue<Customer> {  
    public CustomerLine(int capacity) {  
        super(capacity);  
    }  
    @Override  
    public String toString() {  
        if (this.size() == 0) {  
            return "[Empty]";  
        }  
        StringBuilder sb = new StringBuilder();  
        for (Customer customer : this) {  
            sb.append(customer);  
        }  
        return sb.toString();  
    }  
}
```
  - Генератор клиентов
- ```
public class CustomerGenerator implements Runnable {  
    private CustomerLine customers;  
    private static Random rnd = new Random();  
  
    public CustomerGenerator(CustomerLine customers) {  
        this.customers = customers;  
    }  
  
    @Override  
    public void run() {  
        try {  
            while (!Thread.interrupted()) {  
                TimeUnit.MILLISECONDS.sleep(rnd.nextInt(300)); // клиенты случайны  
                customers.put(new Customer(rnd.nextInt(1000))); // обслуживание случайно  
            }  
        } catch (InterruptedException e) {  
            System.out.println("CustomerGenerator interrupted");  
        }  
        System.out.println("CustomerGenerator finished");  
    }  
}
```

- Кассир обслуживает клиентов

[lesson\\_ch21/ch/ex35/codea](#)

```

public class Teller implements Runnable, Comparable<Teller> {
    private static int counter = 0;
    private final int id = counter++;
    private int customersServed = 0;
    private CustomerLine customers;
    private boolean servingCustomerLine = true;

    public Teller(CustomerLine customers) {
        this.customers = customers;
    }

    public synchronized void doSomethingElse() { // переключиться на другую работу
        customersServed = 0;
        servingCustomerLine = false;
    }

    public synchronized void serveCustomerLine() {
        assert !servingCustomerLine : "already serving: " + this; // выдать сообщение
        servingCustomerLine = true;
        notifyAll(); // выдать данному объекту
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                Customer customer = customers.take(); // взять клиента из очереди
                TimeUnit.MILLISECONDS.sleep(customer.getServiceTime()); // final no sync
                synchronized (this) { // синхронизировано для wait()
                    customersServed++;
                    while (!servingCustomerLine) { // меняется в другом sync методе
                        wait(); // заблокировать данный объект
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(this+ " interrupted");
        }
        System.out.println(this+" terminating");
    }

    @Override // если не Comparable то второй элемент не добавить выдаст Exception
    public synchronized int compareTo(Teller o) {
        return customersServed < o.customersServed ? -1 :
               (customersServed > o.customersServed ? 1 : 0);
    }

    public String shortString() {
        return "T" + id;
    }

    @Override
    public String toString() {
        return "Server " + id;
    }
}

```

- Менеджер кассиров

lesson\_ch21/ch/ex35/codea

```

public class TellerManager implements Runnable {
    private ExecutorService exec;
    private CustomerLine customers; // Server must be Comparable<Server>
    private PriorityQueue<Teller> workingTellers = new PriorityQueue<>();
    private Queue<Teller> tellersDoingOtherThings = new LinkedList<>(); //обычный список
    private int adjustmentPeriod;
    private static Random rnd = new Random();
    public TellerManager(ExecutorService exec, CustomerLine customers,
                         int adjustmentPeriod) {
        this.exec = exec;
        this.customers = customers;
        this.adjustmentPeriod = adjustmentPeriod;
        Teller teller = new Teller(customers); // в начале создается один кассир
        exec.execute(teller);
        workingTellers.add(teller); // в очередь сотрудников
    }
    private void reassignOneTeller() {
        Teller teller = workingTellers.poll(); // изъять из очереди по приоритету
        teller.doSomethingElse(); // переключить на другую работу
        tellersDoingOtherThings.offer(teller); // добавить в очередь другой работы
    }
    public void adjustTellerNumber() { // контроль числа кассиров
        if (customers.size() / workingTellers.size() > 2) { // число клиентов больше
            if (tellersDoingOtherThings.size() > 0) { // если есть неработающие
                Teller teller = tellersDoingOtherThings.remove(); // изъять из списка
                teller.serveCustomerLine(); // поставить его на обслуживание
                workingTellers.offer(teller); // добавить в очередь работающих
                return;
            }
            Teller teller = new Teller(customers); // создать одного
            exec.execute(teller); // запустить его задачу
            workingTellers.offer(teller); // add() { offer() }
            return;
        } // убрать кассира кроме последнего
        if (workingTellers.size() > 1 && customers.size() / workingTellers.size() < 2) {
            reassignOneTeller();
        }
        if (customers.size() == 0) { // клиентов нет
            while (workingTellers.size() > 1) { // выбрать всех кроме одного
                reassignOneTeller();
            }
        }
    }
    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(adjustmentPeriod); // период отработки
                adjustTellerNumber(); // отработать число кассиров
                System.out.print(customers+ " { ");
                for (Teller teller : workingTellers) {
                    System.out.print(teller.shortString()+" ");
                }
                System.out.println(" }");
            }
        } catch (InterruptedException e) {
            System.out.println(this+" interrupted");
        }
        System.out.println(this+" terminating");
    }
    @Override
    public String toString() {
        return "ServerManager ";
    }
}

```

- Применение.

```

public class BankTellerSimulation {
    static final int MAX_LINE_SIZE = 50;
    static final int ADJUSTMENT_PERIOD = 1000; // ms

    public static void check() {
        ExecutorService exec = Executors.newCachedThreadPool();
        CustomerLine customers = new CustomerLine(MAX_LINE_SIZE);
        exec.execute(new CustomerGenerator(customers)); // запустить генератор клиентов
        exec.execute(new TellerManager(exec, customers, ADJUSTMENT_PERIOD));

        try {
            //TimeUnit.MILLISECONDS.sleep(5000);

            System.out.println("Press 'Enter' to quit");
            System.in.read();

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        exec.shutdownNow();
        Time.sleep(100);
    }
}

```

## Моделирование ресторана

- Моделирование ресторана
  - клиенты        выбирают блюдо и заказывают официанту
  - официанты      принимает заказ и отдает его в ресторан очередь заказов
  - повара          берут заказ из очереди и готовят блюдо, передают блюдо официантам
  - официанты      доставляют блюдо клиенту
  - клиент          кушает блюдо, затем заказывает новое, по завершении трапезы уходит
  -
- Пример. реализация работы ресторана просто с заказами от клиентов
  - клиент работает с официантом через заказ по каждому блюду
- **ВНИМАНИЕ.** мощный пример
- Пример. реализация работы ресторана с заказами клиентов по столам
  - клиент занимает место из пула ресторана
  - по месту определяется стол, куда посажен клиент
  - все клиенты за столом одновременно делают заказ, организация через семафоры у клиентов
  - общий заказ стола передается официанту, в очередь ресторана, к шеф повару
  - шеф повар готовит все блюда из заказа по очереди и передает по мере готовности официанту
  - официант доставляет блюдо конкретному клиенту
- **ВНИМАНИЕ.** мощный пример
- 

[lesson\\_ch21/ch/ex35/codea](#)

[lesson\\_ch21/ch/ex36/codea](#)

[lesson\\_ch21/ch/ex36/code](#)

[lesson\\_ch21/ch/ex36/exercise](#)

[lesson\\_ch21/ch/ex36/exercise](#)

## Распределение работы

- Распределение работы [lesson\\_ch21/ch/ex37/codea](#)
- Пример. реализация конвейера по созданию машин
  - все операции выполняются параллельно возможна смена порядка
  - барьер работает параллельно на все операции
  - конвейер состоит из двух очередей, трех роботов и репортера
  -
- Две очереди, carQueue и finishingQueue
  - CarQueue очередь шасси машины
  - FinishinQueue очередь готовы машин
- Роботы EngineRobot, DriveTrainRobot, WheelRobot
  - EngineRobot ожидает запуска, выполняет perform, добавляет себя в pool, и в начало
  - DriveTrainRobot тоже
  - WheelRobot тоже
- RobotPool
  - pool роботов роботы автоматом добавляются при инициализации, и по завершении работы
  - hire(R) извлекает Robot из pool, выполняет R.engage(), выходит
- Assembler
  - сборщик ожидает Car из CarQueue, вызывает все hire(R), кладет результат в finishQueue
- Reporter
  - вывод выбирает машины из очереди FinishingQueue и распечатывает

## Работа CyclicBarier

- Работа CyclicBarier создается с заданным числом тиков [lesson\\_ch21/ch/ex37/codeb](#)
- каждая задача запускает тик барьера и ждет, когда все задачи отстрелялись, вместе продолжают
- используется когда надо не пропустить задачи на следующий этап, например при строительстве
- Два способа работы CyclicBarier с встроенным обработчиком и без него
  - CyclicBarrier с встроенным обработчиком при прохождении барьера запускается обработчик
  - CyclicBarrier без обработчика ничего дополнительно не запускается
- Применение CyclicBarrier
  - один менеджер задач запускает подзадачи до и после барьера обязательно должен быть
  - одна и более подзадач подзадачи которые участвуют в прохождении барьера
  - собственно работа подзадачи и менеджер запускают тики, когда все сработают вместе
  - проходят барьер
  - вполне реально на каждом этапе иметь свой барьер с заданным числом задач
- Пример. реализация конвейера по строительству здания [lesson\\_ch21/ch/ex38/exercise](#)
- реализация поэтапного ожидания
- барьер работает только на одну операцию

## Работа CountDownLatch

- Работа CountDownLatch создается с заданным числом тиков [lesson\\_ch21/ch/ex37/codec](#)
- каждая задача уменьшает счетчик, когда все задачи отстрелялись, тогда продолжается менеджер
- используется когда надо не пропустить задачи на следующий этап, например при строительстве

## Сравнение технологий Mutex

- Сравнение технологий Mutex
  - synchronized применение скрытой блокировки по объекту this
  - Lock применение явной блокировки по объекту Lock
- Пример. реализация сравнения методов блокировки
- ```
public abstract class Incrementable {  
    protected long counter = 0;  
    public abstract void increment();  
}  
  
public class SynchronizingTest extends Incrementable {  
    @Override  
    public synchronized void increment() {  
        counter++; // переменная входит в состав абстрактного класса  
    }  
}  
  
public class LockingTest extends Incrementable {  
    private Lock lock = new ReentrantLock();  
    @Override  
    public void increment() {  
        lock.lock();  
        try {  
            counter++; // переменная входит в состав абстрактного класса  
        } finally {  
            lock.unlock();  
        }  
    }  
}  
  
public class SimpleMicroBenchmark {  
    private static long test(Incrementable incrementable) {  
        long start = System.nanoTime(); // текущее время  
        for (int i = 0; i < 10000000L; i++) {  
            incrementable.increment();  
        }  
        return System.nanoTime() - start; // время затраченное на тест  
    }  
  
    public static void check() {  
        long synchTime = test(new SynchronizingTest());  
        long lockTime = test(new LockingTest());  
        System.out.printf("Synchronized : %10d\n", synchTime);  
        System.out.printf("Lock : %10d\n", lockTime);  
        System.out.printf("Lock/Synchronized:  
%.3f\n", (double)lockTime / (double)synchTime);  
    }  
}
```

[lesson\\_ch21/ch/ex39/codea](#)

## Сравнение технологий Mutex многопоточная реализация

- Сравнение технологий Mutex многопоточная реализация
  - запускается 4 модификатора массива и 4 считывателя, один поток головной
  - все зациклены на барьер, по завершении барьера переход к другому тесту
- Пример. реализация многопоточного теста
- **ВНИМАНИЕ.** Результаты Atomic самый быстрый, затем Lock и самый медленный synchronized
- **ВНИМАНИЕ.** Рекомендуется использовать Lock
- Lock просто в организации, намного проще Atomic и всего в два раза медленнее
- synchronization медленнее Atomic в 5 раз
- 
- 

[lesson\\_ch21/ch/ex39/codeb](#)

## Контейнеры без блокировок

- Контейнеры без блокировок
    - создаются Collections статическими методами
    - стандартные контейнеры все рассинхронизированы
  - Методы
    - CopyOnWriteArrayList создает копию массива, позволяет работать нескольким итераторам
    - не выдает исключение ConcurrentModificationException
    - CopyOnWriteArraySet создает копию для Set
    - ConcurrentHashMap синхронная Map
    - ConCurrentLinkedQueue
  - Пример. реализация CopyOnWriteArrayList [lesson\\_ch21/ch/ex39/codeb](#)
  - используется CyclicBarrier для синхронизации завершения работы над контейнером
  - ```
public class CopyOnWriteListTest {  
    public static void check() {  
        Random rnd = new Random();  
        List<Integer> list = new ArrayList<>();  
        CyclicBarrier barrier = new CyclicBarrier(2);  
  
        // runner  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                Time.sleep(15);  
                List<Integer> listA = new CopyOnWriteArrayList(list);  
                for (int i = 10; i < listA.size(); i++) {  
                    listA.set(i, 25);  
                }  
                try {  
                    barrier.await();  
                } catch (Exception e) {  
                    System.out.println("interrupted");  
                }  
                System.out.println(listA);  
            }  
        }).start();  
  
        // main  
        for (int i = 0; i < 25; i++) {  
            list.add(rnd.nextInt(1000));  
            Time.sleep(1);  
        }  
        try {  
            barrier.await();  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
        System.out.println(list);  
    }  
}
```
  - Пример. реализация получения копии CopyOnWriteArraySet [lesson\\_ch21/ch/ex39/codeb](#)
  - Пример. реализация получения копии ConcurrentHashMap [lesson\\_ch21/ch/ex39/codeb](#)
  - Пример. реализация получения копии ConCurrentLinkedOueue [lesson\\_ch21/ch/ex39/codeb](#)

## **Вопросы производительности**

- Вопросы производительности
  - синхронные контейнеры имеют сильно отличающуюся производительность
  - тестирование позволяет определить какова производительность синхронного контейнера

## **Проверка производительности синхронных ArrayList**

- Проверка производительности синхронных ArrayList
  - проверка synchronizedList() и CopyOnWriteArrayList() для ArrayList
  -
- Пример. реализация теста synchronizedList() и CopyOnWriteArrayList()  
[lesson\\_ch21/ch/ex39/coded](#)
- результаты показывают что CopyOnWriteArrayList() работает быстрее

## **Проверка производительности синхронных ArraySet**

- Проверка производительности синхронных ArraySet
  - он ведет себя точно также как ArrayList в синхронных приложениях

## **Проверка производительности синхронных Map**

- проверка synchronizedMap() и ConcurrentHashMap() для HashMap
- 
- Пример. реализация теста synchronizedMap () и ConcurrentHashMap ()  
[lesson\\_ch21/ch/ex39/codee](#)
- результаты показывают что ConcurrentHashMap () работает быстрее
- 

## **Оптимистическая блокировка**

- Оптимистическая блокировка на базе Atomic классов
  - Atomic проводят вычисления без блокировки mutex, затем используют метод compareAndSet()
  - compareAndSet()      если значение изменилось во время вычислений это ситуация «неудачи»
  - пользователь решает, повторно вычислять или игнорировать битые данные
  - если значение не изменилось, значит «оптимизм» оправдался
- **ВНИМАНИЕ.** Применять только в ограниченных случаях где потеря данных некритична

## Реализация оптимистической блокировки

- Пример. реализация алгоритма «оптимистической блокировки»
  - общий смысл ускорение за счет потери данных
  - обычный способ в 2 раза медленнее на больших объемах данных
- 
- ```
public class FastSimulation {  
    private static final int N_ELEMENTS = 100000;  
    private static final int N_GENES = 30;  
    private static final int N_EVOLVERS = 50; // здоровенный массив 5млн ячеек  
    private static final AtomicInteger[][] GRID =  
        new AtomicInteger[N_ELEMENTS][N_GENES];  
  
    private static Random rnd = new Random();  
    private static class Evolver implements Runnable {  
        @Override  
        public void run() {  
            while (!Thread.interrupted()) {  
                int element = rnd.nextInt(N_ELEMENTS);  
                for (int i = 0; i < N_GENES; i++) {  
                    int previous = element - 1;  
                    if (previous < 0) {  
                        previous = N_ELEMENTS - 1;  
                    }  
                    int next = element + 1;  
                    if (next >= N_ELEMENTS) next = 0;  
                    int oldValue = GRID[element][i].get();  
                    // вычисления  
                    int newValue = oldValue + GRID[previous][i].get() +  
                        GRID[next][i].get();  
                    newValue /= 3;  
                    if (!GRID[element][i].compareAndSet(oldValue, newValue)) {  
                        System.out.println("Old value changed from " + oldValue);  
                    } // здесь обнаруживается ошибка  
                }  
            }  
        }  
    }  
    public static void check() {  
        ExecutorService exec = Executors.newCachedThreadPool();  
        for (int i = 0; i < N_ELEMENTS; i++) {  
            for (int j = 0; j < N_GENES; j++) {  
                GRID[i][j] = new AtomicInteger(rnd.nextInt(1000));  
            }  
        }  
        for (int i = 0; i < N_EVOLVERS; i++) {  
            exec.execute(new Evolver()); // процесс изменяющий базу  
        }  
        try {  
            TimeUnit.MILLISECONDS.sleep(5000);  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
        exec.shutdownNow(); // отключить все потоки  
    }  
}
```
- Пример. реализация сравнения обычной и «оптимистической» блокировок [lesson\\_ch21/ch/ex39/exercise](#)
- **ВНИМАНИЕ.** На больших объемах данных «оптимистическая блокировка» работает в 2 раза быстрее

## ReadWriteLock

- ReadWriteLock оптимизирует ситуацию с редкой записью и частым чтением
  - применять только для оптимизации софта
  - проверять на эффективность в каждом конкретном случае

- **ВНИМАНИЕ.** Неизвестно улучшает ли производительность данный класс, надо проверять
- Пример. реализация ReadWriteLock класса

[lesson\\_ch21/ch/ex40/codea](#)

- создан объект ReadWriteList композиция с ArrayList<>
- внутри используется раздельный Lock на запись и на чтение
- 

- Применение. реализация тестирования

[lesson\\_ch21/ch/ex40/codea](#)

```
public class ReaderWriterListTest {  
    public ExecutorService exec = Executors.newCachedThreadPool();  
    private final static int SIZE = 100;  
    private static Random rnd = new Random();  
    private ReaderWriterList<Integer> list =  
        new ReaderWriterList<>(SIZE, 0); // размер и начальное значение  
    public ReaderWriterListTest(int readers, int writers) {  
        for (int i = 0; i < readers; i++) {  
            exec.execute(new Reader());  
        }  
        for (int i = 0; i < writers; i++) {  
            exec.execute(new Writer());  
        }  
    }  
    private class Writer implements Runnable {  
        @Override  
        public void run() {  
            try {  
                for (int i = 0; i < 20; i++) {  
                    list.set(i, rnd.nextInt());  
                    TimeUnit.MILLISECONDS.sleep(100);  
                }  
            } catch (InterruptedException e) {}  
            System.out.println("Writer finished, shutting down");  
            exec.shutdownNow();  
        }  
    }  
    private class Reader implements Runnable {  
        public void run() {  
            try {  
                for (int i = 0; i < 20; i++) {  
                    list.get(i);  
                    TimeUnit.MILLISECONDS.sleep(1);  
                }  
            } catch (InterruptedException e) {}  
        }  
    }  
    public static void check() {  
        new ReaderWriterListTest(30, 1);  
    }  
}
```

-

- ReaderWriterList < > реализация собственно контейнера с раздельной блокировкой

```

public class ReaderWriterList<T> { // класс List с блокировкой
    private ArrayList<T> lockedList;
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock(true);

    public ReaderWriterList(int size, T initialValue) {
        this.lockedList = new ArrayList<T>(Collections.nCopies(size, initialValue));
    }
    public T set(int index, T element) {
        Lock wlock = lock.writeLock(); // новый writeLock на запись
        wlock.lock();
        try {
            return lockedList.set(index, element);
        } finally {
            wlock.unlock();
        }
    }
    public T get(int index) {
        Lock rlock = lock.readLock(); // новый readLock на запись
        rlock.lock();
        try {
            if (lock.getReadLockCount() > 1) {
                System.out.println(lock.getReadLockCount());
            } // несколько задач получают readLock()

            return lockedList.get(index);
        } finally {
            rlock.unlock();
        }
    }
}

```

## Активные объекты

- Активные объекты модель параллельного программирования
  - каждый объект использует свой собственный поток и очередь сообщений
  - все запросы к объекту ставятся в очередь последовательного выполнения
- Теория
  - создается объект ExecutorService размером в ОДИН поток exec.newSingleThreadExecutor()
  - отправить в очередь объекты Future задач для exec.newSingleThreadExecutor()
  - поставить все объекты Future в синхронный список CopyOnWriteArrayList()
  - все ждать завершения объектов Future<> и получить от них результат Future.get()
- **ВНИМАНИЕ.** Активные объекты используют последовательную обработку сообщений, а не методов

## Правила работы с активными объектами

- Аргументы для активного объекта должны либо
  - быть доступны только для чтения
  - быть другими активными объектами
  - быть объектами не соединенными ни с одной другой задачей
- Свойства активных объектов
  - каждый объект имеет свой рабочий поток
  - каждый объект полностью контролирует доступ к своим полям
  - взаимодействие между активными объектами в форме сообщений между активными объектами
  - все сообщения между активными объектами ставятся в синхронную очередь
- **ВНИМАНИЕ.** Под очередью СООБЩЕНИЙ понимается очередь задач Executors.newSingleThreadExecutor()
- Механизм такой, метод ActiveObject превращается в объект Callable() который помещается в очередь
- И вот этот объект Callable называется «сообщение», каждый метод обязан превращаться в объект.

## Пример. реализация работы активных объектов

- Пример. реализация работы активных объектов

[lesson\\_ch21/ch/ex41/codea](#)

```
public class ActiveObjectDemo {  
    private ExecutorService exec = Executors.newSingleThreadExecutor();  
    private Random rnd = new Random();  
    private void pause(int factor) {  
        try {  
            TimeUnit.MILLISECONDS.sleep(100+rnd.nextInt(factor));  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }  
    // в отдельный поток сразу объект Future  
    public Future<Integer> calculateInt(final int x, final int y) { // объект Future  
        return exec.submit(new Callable<Integer>() {  
            @Override  
            public Integer call() throws Exception {  
                System.out.println("starting "+ x+ " + "+y);  
                pause(50); // 100..150ms  
                return x+y;  
            }  
        });  
    }  
    // в отдельный поток сразу объект Future  
    public Future<Float> calculateFloat(final float x, final float y) { //объект Future  
        return exec.submit(new Callable<Float>() {  
            @Override  
            public Float call() throws Exception {  
                System.out.println("starting "+ x+ " + "+y);  
                pause(200); // 100..200ms  
                return x+y;  
            }  
        });  
    }  
    public void shutdown() {  
        exec.shutdown();  
    }  
    public static void check() {  
        ActiveObjectDemo d1 = new ActiveObjectDemo(); // список с блокировкой синхронный  
        List<Future<?>> results = new CopyOnWriteArrayList<>();  
        for (float f = 0.0f; f < 1.0f; f += 0.2f) {  
            results.add(d1.calculateFloat(f, f)); // выполняемые объекты в разных потоках  
        }  
        for (int i = 0; i < 5; i++) {  
            results.add(d1.calculateInt(i, i));  
        }  
        System.out.println("All asynchronous calls made");  
        while (results.size() > 0) {  
            for (Future<?> f : results) {  
                if (f.isDone()) {  
                    try {  
                        System.out.println(f.get());  
                    } catch (Exception e) {  
                        throw new RuntimeException(e);  
                    }  
                    results.remove(f); // удалить во время итераций CopyOnWriteArrayList  
                }  
            }  
        }  
        d1.shutdown();  
    }  
}
```

## **Выводы по многопоточному программированию**

- Многопоточная программа
  - может выполняться несколько независимых задач
  - разработчик должен проработать завершение задач
  - задачи взаимодействуют через общие ресурсы, чтобы не было конфликтов используют мьютексы
  - в хорошо спроектированной программе взаимная блокировка не возникает
- Преимущества многопоточной программы
  - одновременное выполнение подзадач
  - улучшение структуры кода
  - удобство для конечного пользователя
  - ускоренное переключение контекста
- Недостатки многопоточной программы
  - замедление программы из за блокировок ресурсов
  - дополнительная нагрузка на процессор для управления потоками
  - неоправданное усложнения из за плохого проектирования программы
  - неодинаковое поведение на разных типах компьютеров
- Литература
  - Java Concurrency in Practice, Брайан Гетц, Тим Пейерлс,
  - Concurrent Programming in Java, Дуг Ли

# Graphic User Interface

# Графический интерфейс Swing

- Графический интерфейс Swing альтернатива устаревшей версии AWT
    - библиотека Swing обладает хорошей моделью программирования
    - GUI builders учитывают все аспекты среды разработки
    - простота библиотеки позволяет создавать читаемый код автоматизированными средствами
    - содержит все компоненты графического интерфейса                    кнопки, деревья, таблицы
    - управление с клавиатуры встроено во все компоненты
    - динамический GUI позволяет менять интерфейс на лету
    -

## Основы Swing

- Основы Swing
  - Класс JFrame создание объекта
    - создать объект JFrame задать текст заголовка
    - задать способ закрытия программы
    - задать размеры окна
    - включить видимость
  - Пример. реализация создания окна JFrame

lesson\_ch22/ch/ex01/codea

- **public class** HelloSwing {

```
public static void check() {
    JFrame frame = new JFrame("Hello Swing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // выйти по закрытию окна
    frame.setSize(300, 100); // размер
    frame.setVisible(true); // включить окно
}
```

- Пример. реализация окна с меткой

lesson\_ch22/ch/ex01/codea

- используется прямая передача данных в окно Swing

- **ВНИМАНИЕ.** НЕ РЕКОМЕНДУЕТСЯ

```
public static void check() {
    JFrame frame = new JFrame("Hello Swing");
    JLabel label = new JLabel("A label");
    frame.add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // выйти по закрытию окна
    frame.setSize(300, 100); // размер
    frame.setVisible(true); // включить окно
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    label.setText("Hey! This is Different!");
}
```

- 1

- Работа с потоками Swing
  - управление передается методу с вызовом в отдельном потоке
  - SwingUtilities.invokeLater()      метод постановки задачи в очередь
  - все методы включая main() должны быть подключены через очередь потоков
  - 
  - **ВНИМАНИЕ.** Применение SwingUtilities.invokeLater() позволяет избежать конфликтов
  - **ВНИМАНИЕ.** Нельзя вставлять задержки sleep() в конструктор объекта
  -
- Пример. реализация работы через очередь потоков SwingUtilities.invokeLater() [lesson\\_ch22/ch/ex01/codea](#)
- ```
public class SubmitManipulationTask {
    public static void check() {
        JFrame frame = new JFrame("Hello Swing");
        final JLabel label = new JLabel("A label"); // фиксированное значение
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); //выйти по закрытию окна
        frame.setSize(300, 100); // размер
        frame.setVisible(true); // включить окно
        try {
            TimeUnit.MILLISECONDS.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                label.setText("Hey! This is Different!");
            }
        });
    }
}
```
- Пример. реализация работы через потоки всех объектов [lesson\\_ch22/ch/ex01/codeb](#)
- ```
public class SubmitSwingProgramm extends JFrame {
    private static SubmitSwingProgramm ssp; // этот объект
    private JLabel label;
    public SubmitSwingProgramm() {
        super("Hello SwingM"); // открываем frame new JFrame("Hello")
        label = new JLabel("A label");
        add(label); // frame.add(label)
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setSize(300, 100);
        setVisible(true);
    } // ВНИМАНИЕ сюда задержку вставлять НЕЛЬЗЯ конструктор не сформирует frame
    public static void check() {
        SwingUtilities.invokeLater(new Runnable() { // головной поток
            @Override
            public void run() {
                ssp = new SubmitSwingProgramm(); // просто создали frame в потоке
            }
        });
        try {
            TimeUnit.MILLISECONDS.sleep(1000); //задержку можно ставить в ГОЛОВНОМ ПОТОКЕ
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        SwingUtilities.invokeLater(new Runnable() { // вспомогательный поток
            @Override
            public void run() {
                ssp.label.setText("Hey! This is Different");
            }
        });
    }
}
```

- Пример. реализация случайного числа Label и setLayout()

[lesson\\_ch22/ch/ex02/exercise](#)

```

public class HelloLabel extends JFrame {
    private static List<JLabel> labels;
    private static Random rnd = new Random();

    public HelloLabel() {
        super("Hello Dynamic Label ");
        labels = new CopyOnWriteArrayList<>();

        int nLabel = /*rnd.nextInt(10)*/ +10;
        for (int i = 0; i < nLabel; i++) {
            JLabel label = new JLabel("A label:" + i);
            add(label); // to JFrame
            labels.add(label);
        }
        setLayout(new FlowLayout()); // метод плавающего окна
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // закрыть окно
        setSize(300, 300); // размер
        setVisible(true); // включить окно
    }

    private static class LabelText implements Runnable {
        private String[] strings = new String[]{
            "Hey! This is Different!",
            "It's not the same Color!",
            "Why do you think that dog is smiling?",
            "What the difference between these two balls?",
            "This park is so big!"
        };

        @Override
        public void run() {
            for (int i = 0; i < 3; i++) { // меняем 3 метки всегда
                JLabel label = labels.get(rnd.nextInt(labels.size()));
                String s = strings[rnd.nextInt(strings.length)];
                label.setText(s);
            }
        }
    }

    public static void check() {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new HelloLabel();
            }
        });
    }

    int count = 10;

    try {
        while(count-- > 0) {
            SwingUtilities.invokeLater(new LabelText());
            TimeUnit.MILLISECONDS.sleep(500);
        }
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}

```

## Swing Methods

- JFrame
  - setDefaultCloseOperation(EXIT\_ON\_CLOSE) способ закрытия приложения
  - setSize(300, 100) задать размер окна
  - setVisible(true) включить окно
  - setLayout(new FlowLayout()) метод плавающего окна
  - FlowLayout() плавающее окно
  - BorderLayout() фиксированное окно
  -
- JLabel
  - setText("Text") задать новый текст
  -
- SwingUtilities
  - invokeLater(new Runnable()) запуск объекта в очереди JFrame
  -
- JButton
  - addActionListener() перехватчик событий реализует интерфейс ActionListener
  - actionPerformed() метод определяющий тип события
  - addActionListener() добавить перехватчик события
  - actionPerformed() callback() обработчик события
  - setRolloverEnabled(true) поддержка <Over> курсор мыши над кнопкой
  - setRolloverIcon(faces[1]) Icon при наезде мыши <Over> JButton
  - setPressedIcon(faces[2]) Icon при нажатии JButton
  - setDisabledIcon(faces[4]) Icon при запрете JButton
- JTextField
  - setText() задать текст в поле
  -
- JTextArea
  - append() передает текст в область TextArea
  - setText("") стирает текст в TextArea
- 
- JScrollPane
  - JScrollPane(new JtextArea()) добавляет панель скроллинга для TextArea
  -
-



## Display Framework

- Display Framework
  - SwingConsole вспомогательный класс для работы с JFrame
- Пример. реализация работы с классом SwingConsole
- ```
public class SubmitSwingProgramm extends JFrame {  
    private static SubmitSwingProgramm ssp; // этот объект  
    private JLabel label;  
    public SubmitSwingProgramm() {  
        label = new JLabel("A label");  
        add(label); // frame.add(label)  
    }  
    public static void check() {  
        ssp = new SubmitSwingProgramm();  
        SwingConsole.run(ssp, 300, 100); // головной поток  
        Time.sleep(1000);  
        SwingUtilities.invokeLater(new Runnable() { // вспомогательный поток  
            @Override  
            public void run() {  
                ssp.label.setText("Hey! This is Different");  
            }  
        });  
    }  
}
```

[lesson\\_ch22/ch/ex03/exercise](#)

## Создание кнопки

- Создание кнопки используется класс JButton
  -
- Пример. реализация кнопки на базе класса JButton
- ```
public class Button1 extends JFrame {  
    private JButton b1 = new JButton("Button1");  
    private JButton b2 = new JButton("Button2");  
    public Button1() {  
        setLayout(new FlowLayout());  
        add(b1);  
        add(b2);  
    }  
    public static void check() {  
        SwingConsole.run(new Button1(), 200, 100); // запуск окна  
    }  
}
```

[lesson\\_ch22/ch/ex04/exercise](#)

## Перехват событий

- Перехват событий
  - события называются Event
  - перехватчики событий Listeners «слушают» среду и при появлении события перехватывают его
  - addActionListener() регистрирует обработчик событий
  - actionPerformed() обработчик события

## Реализация Listeners для всех классов

- Реализация Listeners для всех классов
- Пример. реализация addActionListener() для классов библиотеки
  - JButton JMenuItem JCheckBox
  - JComboBox JFileChooser JTextField
  - JTextArea JLabel

[lesson\\_ch22/ch/ex07/exercise](#)

## Текстовое поле класс JTextField

- Текстовое поле класс JTextField
    - класс текстового поля, позволяет выводить текст на экран
  - Пример. реализация Listener и JTextField в привязке к JButton
- [lesson\\_ch22/ch/ex05/codea](#)
- ```
public class Button2 extends JFrame {  
    private JButton b1 = new JButton("Button1");  
    private JButton b2 = new JButton("Button2");  
    private JTextField textField = new JTextField(10); // длина текста  
    class ButtonListener implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            String name = ((JButton) e.getSource()).getText(); // получить текст кнопки  
            textField.setText(name); // по нажатию кнопки вписать ее текст в поле  
        }  
    }  
    private ButtonListener bListener = new ButtonListener(); // один перехватчик  
    public Button2() {  
        b1.addActionListener(bListener); // перехватчик событий один и тот же  
        b2.addActionListener(bListener);  
        setLayout(new FlowLayout());  
        add(b1);  
        add(b2);  
        add(textField);  
    }  
    public static void check() {  
        SwingConsole.run(new Button2(), 200, 150);  
    }  
}
```

## Текстовые области класс JTextArea

- Текстовые области – это поля многострочного текста
  - Пример. реализация работы с JTextArea
- [lesson\\_ch22/ch/ex06/codea](#)
- ```
public class TextArea extends JFrame {  
    private JButton b1 = new JButton("Add Data");  
    private JButton c1 = new JButton("Clear Data");  
    private JTextArea textArea = new JTextArea(20, 40); // текстовое поле  
    private Map<String, String> map = new HashMap<>();  
  
    public TextArea() { // как сработает Listener по кнопке b1 влепит всю Map в поле  
        map.putAll(Countries.capitals()); // все столицы поместить в Map  
        b1.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                for (Map.Entry<String, String> entry : map.entrySet()) {  
                    textArea.append(entry.getKey() + " : " + entry.getValue() + "\n");  
                }  
            }  
        });  
        c1.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                textArea.setText(""); // стереть все поле  
            }  
        });  
        setLayout(new FlowLayout());  
        add(new JScrollPane(textArea)); // добавить панель со скроллингом  
        add(b1); // кнопка добавления текста  
        add(c1); // кнопка стирания текста  
    }  
    public static void check() {  
        SwingConsole.run(new TextArea(), 475, 425);  
    }  
}
```

## Управление расположением компонентов

- Управление расположением компонентов
  - размещение компонентов производится прямо в программе
  - размещение определяется менеджером размещения, а не координатами
  - Классы Container JApplet, JFrame, JWindow, JPanel могут содержать Component
  - Container.setLayout() позволяет выбрать менеджера размещения

[lesson\\_ch22/ch/ex09/codea](#)

## BorderLayout

- BorderLayout менеджер размещения используемый по умолчанию
  - компонент размещается в центре объекта по умолчанию, либо по границам окна
- Четыре области размещения
  - BorderLayout.NORTH вдоль верхней границы окна
  - BorderLayout.SOUTH вдоль нижней границы окна
  - BorderLayout.EAST вдоль правой стороны окна
  - BorderLayout.WEST вдоль левой стороны окна
  - BorderLayout.CENTER размещение по центру окна
- Пример. реализация размещения с BorderLayout <BorderLayout1>
- ```
public class BorderLayout1 extends JFrame {
    public BorderLayout1() {
        add(BorderLayout.NORTH, new JButton("North"));
        add(BorderLayout.SOUTH, new JButton("South"));
        add(BorderLayout.EAST, new JButton("East"));
        add(BorderLayout.WEST, new JButton("West"));
        add(BorderLayout.CENTER, new JButton("Center"));
    }

    public static void check() {
        SwingConsole.run(new BorderLayout1(), 300, 250);
    }
}
```

[lesson\\_ch22/ch/ex09/codea](#)

## FlowLayout

- FlowLayout менеджер размещения
  - компоненты размещаются один за другим, слева направо, сверху вниз
  - размер компонентов сжимается до минимально возможного
  - компоненты размещаются адаптивно подстраиваясь под размер окна и расположение друг друга
  -
- Пример. реализация размещения с FlowLayout <FlowLayout1>
- ```
public class FlowLayout1 extends JFrame {
    public FlowLayout1() {
        setLayout(new FlowLayout());
        for (int i = 0; i < 4; i++) {
            add(new JButton("Button " + i));
        }
    }

    public static void check() {
        SwingConsole.run(new FlowLayout1(), 300, 300);
    }
}
```

[lesson\\_ch22/ch/ex09/codea](#)

## GridLayout

- GridLayout менеджер размещения
  - компоненты размещаются в по сетке один за другим, слева направо, сверху вниз
  - при использовании надо указать размеры таблицы, число строк и столбцов
- **ВНИМАНИЕ.** Для размещения объекта в нужной ячейке можно использовать JPanel() объект
- Пример. реализация размещения с GridLayout < GridLayout 1> [lesson\\_ch22/ch/ex09/codea](#)
- ```
public class GridLayout1 extends JFrame {
    public GridLayout1() {
        setLayout(new GridLayout(7, 3));
        for (int i = 0; i < 20; i++) {
            add(new JButton("Button " + i));
        }
    }

    public static void check() {
        SwingConsole.run(new GridLayout1(), 300, 300);
    }
}
```
- Пример. реализация размещения объектов в ячейках GridLayout [lesson\\_ch22/ch/ex17/exercise](#)

## GridBagLayout

- GridBagLayout менеджер размещения
  - компоненты размещаются в ручном режиме, можно указать подробно размещение
  - за счет ручного размещения это самый сложный менеджер размещения

## Абсолютное позиционирование

- Абсолютное позиционирование размещение объекта в позиции в пикселях
  - для абсолютного позиционирования надо сделать следующее
  - вызвать setLayout(null) то есть отказаться от менеджера размещения
  - для каждого компонента setBounds() или reshape() передать размеры компонента в пикселях
  - можно сделать в конструкторе компонента или методе paint()
  -

## BoxLayout

- BoxLayout менеджер размещения, развитие GridBagLayout, не такой сложный
  - компоненты размещаются в ручном режиме, но более просто, чем с GridBagLayout

## Ручное или Автоматическое размещение

- для простых GUI подходит ручное размещение
- для сложных GUI подходит автоматическое размещение
- **ВНИМАНИЕ.** для сложных интерфейсов применять ТОЛЬКО автоматическое размещение
-

## Модель событий библиотеки Swing

- Модель событий библиотеки Swing
  - любой компонент может создавать Event
  - любое событие может быть перехвачено Listeners
  - регистрация Listeners – вызов метода класса объекта вида `addNnnnListener()`
- ВНИМАНИЕ. Чтобы не искать поддерживаемые Listeners использовать программу Reflections

[lesson\\_ch22/ch/ex09/codeb](#)

## Типы Events и Listeners

- Типы Events и Listeners

| Event, Listener                                                                                                          | Components                                                                                                                                                                            | Note                   |
|--------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|
| ActionEvent<br>ActionListener<br><code>addActionListener()</code><br><code>removeActionListener()</code>                 | JButton, JList, JTextField, JMenuItem<br>JCheckBoxMenuItem,<br>JMenu,<br>JPopupMenu                                                                                                   |                        |
| AdjustmentEvent<br>AdjustmentListener<br><code>addAdjustmentListener()</code><br><code>removeAdjustmentListener()</code> | JScrollbar,<br>Ajustable Interface                                                                                                                                                    |                        |
| ComponentEvent<br>ComponentListener<br><code>addComponentListener()</code><br><code>removeComponentListener()</code>     | Component*<br>JButton, JCheckBox, JComboBox, Container,<br>JPanel, JApplet, JScrollPane, Window, JDialog,<br>JFileDialog, JFrame, JLabel, JList, JScrollbar,<br>JTextArea, JTextField |                        |
| ContainerEvent<br>ContainerListener<br><code>addContainerListener()</code><br><code>removeContainerListener()</code>     | Container *<br>JPanel, JApplet, JScrollPane, Window, Idialog,<br>DFileDialog, IFrame                                                                                                  |                        |
| FocusEvent FocusListener<br><code>addFocusListener()</code><br><code>removeFocusListener()</code>                        | Component*                                                                                                                                                                            | * означает все ПОТОМКИ |
| KeyEvent KeyListener<br><code>addKeyListener()</code><br><code>removeKeyListener()</code>                                | Component*                                                                                                                                                                            |                        |
| MouseEvent MouseListener<br><code>addMouseListener()</code><br><code>removeMouseListenerQ</code>                         | Component*                                                                                                                                                                            |                        |
| MouseEvent<br>MouseMotionListener<br><code>addMouseMotionListener()</code><br><code>removeMouseMotionListenerQ</code>    | Component*                                                                                                                                                                            |                        |
| WindowEvent WindowListener<br><code>addWindowListener()</code><br><code>removeWindowListenerQ</code>                     | Window*<br>UDialog, DFileDialog и DFrame                                                                                                                                              |                        |
| ItemEvent ItemListener<br><code>addItemListener()</code><br><code>removeItemListener()</code>                            | DCheckBox, DCheckBoxMenuItem, JComboBox,<br>TList<br>ItemSelectable Interface                                                                                                         |                        |
| TextEvent TextListener<br><code>addTextListener()</code><br><code>removeTextListener()</code>                            | JTextComponent<br>JTextArea, DTextField                                                                                                                                               |                        |
|                                                                                                                          |                                                                                                                                                                                       |                        |

- Пример. реализация Reflections для объектов Swing

[lesson\\_ch22/ch/ex09/codeb](#)

```

• public class ShowAddListeners extends JFrame {
    private JTextField nameField = new JTextField(25); // поле метода
    private JTextArea results = new JTextArea(40, 65); // поле результата
    private static Pattern pattern = Pattern.compile("(add\\w+?Listener\\(.+?)\\))");
    private static Pattern qualifier = Pattern.compile("\\w+\\.");
    class NameL implements ActionListener { // обработчик сообщений
        @Override
        public void actionPerformed(ActionEvent e) {
            String nameClass = nameField.getText().trim(); // сжать пробелы
            if (nameClass.length() == 0) {
                results.setText("No match"); // не нашли
                return;
            }
            Class<?> kind;
            try {
                kind = Class.forName("javax.swing." + nameClass);
            } catch (ClassNotFoundException ex) {

                results.append("No match\n"); // нет такого класса
                return;
            }
            results.setText(""); // сбросить текст окна перед отработкой методов
            for (Method method : kind.getMethods()) {
                Matcher matcher = pattern.matcher(method.toString()); // имена методов
                if (matcher.find()) {
                    String methodName = matcher.group(); // замещает группы <fssdf.>
                    results.append(qualifier.matcher(methodName).replaceAll("") + "\n");
                }
            }
        }
    }
    public ShowAddListeners() {
        NameL nameListener = new NameL();
        nameField.addActionListener(nameListener); // добавили в поле обработчик по Enter
        JPanel top = new JPanel();
        top.add(new JLabel("Swing class name (press Enter):"));
        top.add(nameField);
        add(BorderLayout.NORTH, top);
        add(new JScrollPane(results)); // textArea в панель по центру
        nameListener.actionPerformed(new ActionEvent("", 0, ""));
    }
}
}

```

## Создание обработчиков событий

- Создание обработчиков событий на базе информации о событии
  - взять имя события который поддерживает класс объекта Swing
  - заменить слово Event на Listener FocusEvent => FocusListener Interface
  - добавить add для метода FocusListener => addFocusListener() method
  - new FocusListener {} объект Listener
  - addFocusListener() callback Listener

## Адаптеры

- Адаптеры это интерфейсы которые позволяет переопределять только нужный метод
  - остальные методы уже созданы в Адаптере в виде пустышек
- **ВНИМАНИЕ.** Использовать @Override для проверки НУЖНОГО МЕТОДА

## Интерфейсы Listener

- Таблица интерфейсов Listener

| Interface                                 | Methods                                                                                                                                                                                                                  | Note |
|-------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| ActionListener                            | actionPerformed(ActionEvent)                                                                                                                                                                                             |      |
| AdjustmentListener                        | adjustmentValueChanged(AdjustmentEvent)                                                                                                                                                                                  |      |
| ComponentListener<br>ComponentAdapter     | componentHidden(ComponentEvent)<br>componentShown(ComponentEvent)<br>componentMoved(ComponentEvent)<br>componentResized(ComponentEvent)                                                                                  |      |
| ContainerListener<br>ContainerAdapter     | componentAdded(ContainerEvent)<br>componentRemoved(ContainerEvent)                                                                                                                                                       |      |
| FocusListener<br>FocusAdapter             | focusGained(FocusEvent)<br>focusLost( FocusEvent)                                                                                                                                                                        |      |
| KeyListener<br>KeyAdapter                 | keyPressed(KeyEvent)<br>keyReleased(KeyEvent)<br>keyTyped(KeyEvent)                                                                                                                                                      |      |
| MouseListener<br>MouseAdapter             | mouseClicked(MouseEvent)<br>mouseEntered(MouseEvent)<br>mouseExited(MouseEvent)<br>mousePressed(MouseEvent)<br>mouseReleased(MouseEvent)                                                                                 |      |
| MouseMotionListener<br>MouseMotionAdapter | mouseDragged(MouseEvent)<br>mouseMoved(MouseEvent)                                                                                                                                                                       |      |
| WindowListener<br>WindowAdapter           | windowOpened(WindowEvent)<br>windowClosing(WindowEvent)<br>windowClosed(WindowEvent)<br>windowActivated(WindowEvent)<br>windowDeactivated(WindowEvent)<br>windowIconified(WindowEvent)<br>windowDeiconified(WindowEvent) |      |
| ItemListener                              | itemStateChanged(ItemEvent)                                                                                                                                                                                              |      |

•

## Отслеживание нескольких событий

- Отслеживание нескольких событий
  - Пример. реализация отслеживания нескольких событий на одной JButton
- [lesson\\_ch22/ch/ex10/codea](#)
- ```
public class TrackEvent extends JFrame {  
    private HashMap<String, JTextField> hMap = new HashMap<>();  
    private String[] events = {  
        "focusGained", "focusLost", "keyPressed",  
        "keyReleased", "keyTyped", "mouseClicked",  
        "mouseEntered", "mouseExited", "mousePressed",  
        "mouseReleased", "mouseDragged", "mouseMoved"  
    };  
    class MyButton extends JButton{  
        void report(String field, String msg) {  
            hMap.get(field).setText(msg);  
        }  
        FocusListener f1 = new FocusListener() {  
            public void focusGained(FocusEvent e) {  
                report("focusGained",e paramString()); // показывает что сработало  
            }  
            public void focusLost(FocusEvent e) {  
                report("focusLost",e paramString()); // просто показывает что сработало  
            }  
        };  
        KeyListener k1 = new KeyListener() {  
            public void keyTyped(KeyEvent e) {  
                report("keyTyped",e paramString()); // просто показывает что сработало  
            }  
            public void keyPressed(KeyEvent e) {  
                report("keyPressed",e paramString()); // просто показывает что сработало  
            }  
            public void keyReleased(KeyEvent e) {  
                report("keyReleased",e paramString()); // показывает что сработало  
            }  
        };  
        MouseListener m1 = new MouseListener() {  
            public void mouseClicked(MouseEvent e) {  
                report("mouseClicked",e paramString()); // показывает что сработало  
            }  
            public void mousePressed(MouseEvent e) {  
                report("mousePressed",e paramString()); // показывает что сработало  
            }  
            public void mouseReleased(MouseEvent e) {  
                report("mouseReleased",e paramString()); // показывает что сработало  
            }  
            public void mouseEntered(MouseEvent e) {  
                report("mouseEntered",e paramString()); // показывает что сработало  
            }  
            public void mouseExited(MouseEvent e) {  
                report("mouseExited",e paramString()); // показывает что сработало  
            }  
        };  
        MouseMotionListener mm1 = new MouseMotionListener() {  
            public void mouseDragged(MouseEvent e) {  
                report("mouseDragged",e paramString()); // показывает что сработало  
            }  
            @Override  
            public void mouseMoved(MouseEvent e) {  
                report("mouseMoved",e paramString()); // просто показывает что сработало  
            }  
        };  
    };
```

- Продолжение. реализация отслеживания нескольких событий на одной JButton [lesson\\_ch22/ch/ex10/codea](#)
- ```
public MyButton(Color color, String label) { // это setup кнопки
    super(label);
    setBackground(color);
    addFocusListener(f1);
    addKeyListener(k1);
    addMouseListener(ml);
    addMouseMotionListener(mml);
}
}

private MyButton b1 = new MyButton(Color.BLUE, "test1");
private MyButton b2 = new MyButton(Color.RED, "test2");

public TrackEvent() {
    setLayout(new GridLayout(events.length+1,2)); // целая туча событий
    for (String event : events) {
        JTextField textField = new JTextField(); // текстовое поле
        textField.setEditable(true);
        add(new JLabel(event,JLabel.RIGHT)); // добавить надпись к полю
        add(textField); // добавить само поле
        hMap.put(event,textField); // в карту для отработки Listener
    }
    add(b1);
    add(b2);
}

public static void check() {
    SwingConsole.run(new TrackEvent(),700,500); // размещение по сетке
}
```
- Пример. реализация двух событий JButton и перевод нажатий в TextField [lesson\\_ch22/ch/ex10/exercise](#)
- ```
public class ButtonFocus extends JFrame {
    JButton button = new JButton("Click and Type");
    JTextField textField = new JTextField();
    JLabel label = new JLabel("<Please click on button type the text>", JLabel.LEFT);
    JLabel label2 = new JLabel("Input text(click on button to erase):", JLabel.LEFT);
    class BMListener extends MouseAdapter {
        @Override
        public void mouseClicked(MouseEvent e) {
            textField.setText("");
        }
    }
    class BKListener extends KeyAdapter {
        @Override
        public void keyTyped(KeyEvent e) {
            textField.setText(textField.getText()+e.getKeyChar());
        }
    }
    public ButtonFocus() {
        setLayout(new GridLayout(5, 1)); // целая туча событий
        button.addMouseListener(new BMListener()); // FocusListener
        button.addKeyListener(new BKListener());
        button.setBackground(Color.CYAN);
        textField.setEditable(true);
        add(label);
        add(button);
        add(label2);
        add(textField);
    }
    public static void check() {
        SwingConsole.run(new ButtonFocus(), 400, 200); // размещение по сетке
    }
}
```

## Компоненты Swing

### Buttons

- Buttons
  - библиотека включает несколько видов кнопок
  - все кнопки производные от класса AbstractButton{}
- Виды кнопок
  - JButton new JButton("JButton");
  - BasicArrowButton new BasicArrowButton(BasicArrowButton.NORTH); // кнопка стрелки вверх
  - JToggleButton new JToggleButton("JToggleButton");
  - JCheckBox checkBox new JCheckBox("JCheckBox");
  - JRadioButton new JRadioButton("JRadioButton");
- Пример. реализация Buttons в Swing lesson\_ch22/ch/ex13/codea

### Группы кнопок

- Группы кнопок
    - используются для создания взаимоисключающих состояний переключателей
    - строятся на всех типах кнопок от AbstractButton
    - создается группа кнопок, затем каждая кнопка добавляется в группу и как обычно на панель
  - Пример. реализация Group Buttons в Swing lesson\_ch22/ch/ex13/codeb
- ```
public class ButtonGroups extends JFrame {  
    private static String[] ids = {  
        "June", "Ward", "Beaver", "Wally", "Eddie", "Lumpy"  
    };  
  
    static JPanel makeBPanel(Class<? extends AbstractButton> kind, String[] ids) {  
        ButtonGroup bg = new ButtonGroup();  
        JPanel jp = new JPanel();  
        String title = kind.getName(); // вытащить заключительный текст кнопки  
        title = title.substring((title.lastIndexOf(".")) + 1)); // вытащить текст кнопки  
        jp.setBorder(new TitledBorder(title));  
        for (String id : ids) {  
            AbstractButton ab = new JButton("failed");  
            try {  
                Constructor ctor = kind.getConstructor(String.class);  
                ab = (AbstractButton) ctor.newInstance(id);  
            } catch (Exception e) { // создать методом Reflections кнопку нужного класса  
                System.out.println("Can't create " + kind);  
            }  
            bg.add(ab); // в группу bg добавить кнопку  
            jp.add(ab); // эту же кнопку добавить в панель  
        }  
        return jp; // вернуть панель  
    }  
    public ButtonGroups() {  
        setLayout(new FlowLayout());  
        add(makeBPanel(JButton.class,ids));  
        add(makeBPanel(JToggleButton.class,ids));  
        add(makeBPanel(JCheckBox.class,ids));  
        add(makeBPanel(JRadioButton.class,ids));  
    }  
  
    public static void check() {  
        SwingConsole.run(new ButtonGroups(), 500, 350);  
    }  
}
```

## Icon

- Icon значки
    - используются внутри JLabel или в компонентах унаследованных от AbstractButton
    - JButton, JCheckBox, JRadioButton или JMenuItem
  - Картинки для значков
    - любые файлы формата GIF загружаются через ImageIcon объект
  - Пример. реализация работы с Icon
- [lesson\\_ch22/ch/ex13/codec](#)
- ```
public class Faces extends JFrame {  
    private static Icon[] faces;  
    private JButton jb = new JButton("Disable");  
    private JButton jb2 = new JButton("Disable");  
    private boolean mad = false;  
    public Faces() {  
        // URL loc = getClass().getProtectionDomain().getCodeSource().getLocation();  
        // URL imageURL = new URL(loc+"/"+ "images/Face0.gif");  
        URL loc = getClass().getResource("images/Face20.gif");  
        faces = new ImageIcon[]{  
            new ImageIcon(getClass().getResource("images/Face0.gif")),  
            new ImageIcon(getClass().getResource("images/Face1.gif")),  
            new ImageIcon(getClass().getResource("images/Face2.gif")),  
            new ImageIcon(getClass().getResource("images/Face3.gif")),  
            new ImageIcon(getClass().getResource("images/Face4.gif")),  
        };  
        jb = new JButton("JButton", faces[3]);  
        setLayout(new FlowLayout()); //layout  
        jb.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                if (mad) {  
                    jb.setIcon(faces[3]);  
                    mad = false;  
                } else {  
                    jb.setIcon(faces[0]);  
                    mad = true;  
                }  
                jb.setVerticalAlignment(JButton.TOP);  
                jb.setHorizontalTextPosition(JButton.LEFT);  
            }  
        });  
        jb.setRolloverEnabled(true); // Rollover Support поддержка <Over>  
        jb.setRolloverIcon(faces[1]); // rollover icons при наезде мыши <Over>  
        jb.setPressedIcon(faces[2]); // при нажатии  
        jb.setDisabledIcon(faces[4]); // при запрете  
        jb.setToolTipText("Wow!"); // toolTipText текст появляется через 3 сек <Over>  
        add(jb);  
        jb2.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                if (jb.isEnabled()) {  
                    jb.setEnabled(false);  
                    jb2.setText("Enable"); // походу одна кнопка контролирует другую  
                } else {  
                    jb.setEnabled(true);  
                    jb2.setText("Disable"); // походу одна кнопка контролирует другую  
                }  
            }  
        });  
        add(jb2);  
    }  
    public static void check() {  
        SwingConsole.run(new Faces(), 250, 125); // FlowLayout()  
    }  
}
```

## JTooltip

- JTooltip подсказки текст, появляется при наведении курсора мыши на объект через 3 сек
  - setToolTipText(«text») задать текст подсказки для объекта
  - поле показывает текст через несколько секунд после наведения курсора на объект
- Пример. реализация setToolTipText() [lesson\\_ch22/ch/ex13/codec](#)

## Borders

- Borders рамки это окантовка компонентов Swing
  - практически все компоненты производные от JComponent могут менять Border
- Пример. реализация setBorder() для различных компонентов [lesson\\_ch22/ch/ex14/codea](#)

```
• public class Borders extends JFrame{
    public static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String name = b.getClass().toString(); // имя класса в строку
        name = name.substring(name.lastIndexOf(".") + 1); // вытащить чистое имя
        jp.add(new JLabel(name, JLabel.CENTER), BorderLayout.CENTER); // в центр
        jp.setBorder(b); // задать новый Border
        return jp; // вернуть новую панель
    }
    public Borders() {
        setLayout(new GridLayout(2, 4)); // 2 на 4 поле
        add(showBorder(new TitledBorder("Title"))); // добавить панель с именем класса
        add(showBorder(new EtchedBorder()));
        add(showBorder(new LineBorder(Color.BLUE)));
        add(showBorder(new MatteBorder(5, 5, 30, 30, Color.GREEN)));
        add(showBorder(new BevelBorder(BevelBorder.RAISED)));
        add(showBorder(new SoftBevelBorder(BevelBorder.LOWERED)));
        add(showBorder(new CompoundBorder(new EtchedBorder(),
                                         new LineBorder(Color.RED))));
    }
    public static void check() {
        SwingConsole.run(new Borders(), 500, 300);
    }
}
```

## JtextField

- JTextField текстовые поля
- Подключение документа JPlainDocument к JTextField
  - при подключении документа, обработка ввода производится в этом документе
  - на каждый введенный символ производится обработка он ничего не накапливает по умолчанию
  - управление документу передается РАНЬШЕ чем Listener поля JTextField
- Пример. реализация JTextField с подключенным документом JPlainDocument [lesson\\_ch22/ch/ex13/coded](#)
- **ВНИМАНИЕ.** Можно подключить документ к другому, два поля с документами, заполнять оба через поле

```
• public class UpperCaseDocument extends PlainDocument { // документ
    private boolean upperCase = true;
    public void setUpperCase(boolean flag) {
        upperCase = flag;
    }
    @Override // вставить строку если надо перевести в UpperCase
    public void insertString(int offset, String str,
                           AttributeSet attSet) throws BadLocationException {
        if (upperCase) {
            str = str.toUpperCase();
        }
        super.insertString(offset, str, attSet);
    }
}
```

- Применение. класс TextFields

lesson\_ch22/ch/ex13/coded

```

• public class TextFields extends JFrame {
    private JButton b1 = new JButton("Get Text");
    private JButton b2 = new JButton("Set Text");
    private JTextField t1 = new JTextField(30);
    private JTextField t2 = new JTextField(30);
    private JTextField t3 = new JTextField(30);
    private String s = "";
    private UpperCaseDocument ucd = new UpperCaseDocument();
    class T1 implements DocumentListener {
        public void insertUpdate(DocumentEvent e) {
            t2.setText(t1.getText()); // вставить в поле t2 текст из t1
            t3.setText("Text: " + t1.getText()); // тоже но с надписью
        }
        public void removeUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
        }
        public void changedUpdate(DocumentEvent e) {
        }
    }
    class T1A implements ActionListener {
        private int count = 0;
        @Override
        public void actionPerformed(ActionEvent e) {
            t3.setText("t1 Action Event " + count++); // подсчитывает число событий t1
        }
    }
    class B1 implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (t1.getSelectedText() == null) {
                s = t1.getText(); // полный текст если нет выделения
            } else {
                s = t1.getSelectedText(); // выделение
            }
            t1.setEditable(true); // открыть t1 для редактирования
        }
    }
    class B2 implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            ucd.setUpperCase(false); // запретить перевод в UpperCase
            t1.setText("Inserted by Button2: " + s);
            ucd.setUpperCase(true); // восстановить UpperCase
            t1.setEditable(false); // запрет редактирования t1 пока не нажата b1
        }
    }
    public TextFields() {
        t1.setDocument(ucd); // подключено к документу ucd
        ucd.addDocumentListener(new T1());
        b1.addActionListener(new B1()); // читает строку s из t1
        b2.addActionListener(new B2()); // пишет s в t1 без UpperCase по b2 запрет t1
        t1.addActionListener(new T1A()); // пишет счетчик изменений t1
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(t1);
        add(t2);
        add(t3);
    }
    public static void check() {
        SwingConsole.run(new TextFields(), 375, 200);
    }
}

```

## JPasswordField

- JPasswordField это поле ввода пароля
  - для него можно задать символ скрывающий символы
  - requestFocus() метод смены фокуса на данное поле
- **ВНИМАНИЕ.** Перевод фокуса делается через метод requestFocus()
- Пример. реализация JPasswordField с поддержкой стирания прежнего текста [lesson\\_ch22/ch/ex17/exercise](#)
  - стирание сделано на базе KeyListener который стирает старый текст по нажатию первого символа
  - реализация размещения объектов в ячейках GridLayout
- ```
public class PasswordMessage extends JFrame {  
    private JButton[] buttons = {  
        new JButton("Alert"), new JButton("Yes/No"),  
        new JButton("Color"), new JButton("Input"),  
        new JButton("3 Vals")  
    };  
    private JTextField textField = new JTextField(30);  
    private JPasswordField passwordField = new JPasswordField("Password", 30);  
    private String stringPassword = "111111";  
    private JTextArea textArea = new JTextArea(200, 50);  
    private JPanel[] jPanels = { //place Holders  
        new JPanel(), new JPanel(), new JPanel(),  
        new JPanel(), new JPanel()  
    };  
    private boolean focusFirst = false;  
    public PasswordMessage() {  
        passwordField.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                String s = new String(passwordField.getPassword());  
                if (s.equals(stringPassword)) {  
                    JOptionPane.showMessageDialog(null,  
                        "Access granted!", "System Message",  
                        JOptionPane.INFORMATION_MESSAGE);  
                } else {  
                    JOptionPane.showMessageDialog(null,  
                        "Access denied!", "System Message",  
                        JOptionPane.ERROR_MESSAGE);  
                }  
                textField.requestFocus(); // focus on textField  
                textField.setText(s);  
                focusFirst = false; // focus lost  
            }  
        });  
        passwordField.addKeyListener(new KeyAdapter() {  
            @Override  
            public void keyTyped(KeyEvent e) {  
                if (!focusFirst) {  
                    passwordField.setText(""); // стереть и добавить первый символ  
                    focusFirst = true;  
                }  
            }  
        });  
        setLayout(new GridLayout(5, 1));  
        jPanels[0].add(new JLabel("Please input password:"));  
        jPanels[1].add(passwordField);  
        jPanels[4].add(textField); // at last position  
        for (JPanel jPanel : jPanels) {  
            add(jPanel);  
        }  
    }  
    public static void check() {  
        SwingConsole.run(new PasswordMessage(), 400, 240);  
    }  
}
```

## Мини редактор

- JTextPane управляющий элемент, может редактировать тексты

[lesson\\_ch22/ch/ex14/codeb](#)

```
public class TextPane extends JFrame {
    private JButton b = new JButton("Add Text");
    private JTextPane tp = new JTextPane();
    private static IGenerator sg = new GenRnd.GenStr(7);
    public TextPane() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for (int i = 0; i < 10; i++) {
                    tp.setText(tp.getText() + sg.next() + "\n"); // по нажатию по 7 символов
                }
            }
        });
        add(new JScrollPane(tp));
        add(BorderLayout.SOUTH, b); // кнопку вниз окна
    }
    public static void check() {
        SwingConsole.run(new TextPane(), 475, 425);
    }
}
```

- Пример. реализация редактирования текстов JTextArea

[lesson\\_ch22/ch/ex14/exercise](#)

## JCheckBox

- JCheckBox флагки объекты меню с двумя состояниями «включено/выключено»
  - есть два типа флагков JCheckBox и JRadioButton
  - создаются конструктором с текстом надписи
  - с событиями работает точно также как JButton

- Пример. реализация работы с JCheckBox

[lesson\\_ch22/ch/ex15/codea](#)

```
public class CheckBoxes extends JFrame {
    private JTextArea textArea = new JTextArea(6, 15);
    private JCheckBox cb1 = new JCheckBox("Check Box 1");
    private JCheckBox cb2 = new JCheckBox("Check Box 2");
    public CheckBoxes() {
        cb1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("1", cb1);
            }
        });
        cb2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("2", cb2);
            }
        });
        setLayout(new FlowLayout());
        add(new JScrollPane(textArea));
        add(cb1);
        add(cb2);
    }
    public void trace(String b, JCheckBox cb) {
        if (cb.isSelected()) {
            textArea.append("Box " + b + " Set\n");
        } else {
            textArea.append("Box " + b + " Cleared\n");
        }
    }
    public static void check() {
        SwingConsole.run(new CheckBoxes(), 200, 300);
    }
}
```

## JRadioButton

- JRadioButton флагки используемые как правило как переключатели в группе
  - для создания переключателя надо объединить флагки в группу
- Пример. реализация JRadioButton переключателя
- ```
public class RadioButtons extends JFrame{  
    private JTextField textField = new JTextField(15);  
    private ButtonGroup buttonGroup = new ButtonGroup();  
    private JRadioButton rb1 = new JRadioButton("one", false);  
    private JRadioButton rb2 = new JRadioButton("two", false);  
    private JRadioButton rb3 = new JRadioButton("three", false);  
    private ActionListener al = new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            textField.setText("Radio button "+((JRadioButton)e.getSource()).getText());  
        }  
    };  
    public RadioButtons() throws HeadlessException {  
        rb1.addActionListener(al);  
        rb2.addActionListener(al);  
        rb3.addActionListener(al);  
        buttonGroup.add(rb1); // группа только для логики работы  
        buttonGroup.add(rb2);  
        buttonGroup.add(rb3);  
        setLayout(new FlowLayout());  
        add(textField);  
        add(rb1);  
        add(rb2);  
        add(rb3);  
    }  
    public static void check() {  
        SwingConsole.run(new RadioButtons(), 200, 125);  
    }  
}
```

[lesson\\_ch22/ch/ex16/codea](#)

## JTabbedPane

- JTabbedPane Панель вкладок позволяет создать окно с набором вкладок
- Пример. реализация JTabbedPane

[lesson\\_ch22/ch/ex17/codea](#)

```
public class TabbedPanel1 extends JFrame {  
    private String[] flavors = {  
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",  
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",  
        "Praline Cream", "Mud Pie"  
    };  
    private JTabbedPane tabbedPane = new JTabbedPane();  
    private JTextField textField = new JTextField(20);  
    public TabbedPanel1() {  
        int i = 0;  
        for (String flavor : flavors) {  
            tabbedPane.addTab(flavor, new JButton("Tabbed pane " + i++));  
        }  
        tabbedPane.addChangeListener(new ChangeListener() {  
            @Override  
            public void stateChanged(ChangeEvent e) {  
                textField.setText("Tab selected: "+tabbedPane.getSelectedIndex());  
            }  
        });  
        add(BorderLayout.SOUTH, textField);  
        add(tabbedPane);  
    }  
    public static void check() {  
        SwingConsole.run(new TabbedPanel1(), 400, 250);  
    }  
}
```

## JComboBox

- JComboBox раскрывающиеся списки, занимает одну строку, при раскрытии показывает весь список
  - позволяет выбрать элемент из выпадающего списка фиксированного
  - позволяет ввести собственный элемент если вызван метод setEditable()

- Пример. реализация JComboBox

[lesson\\_ch22/ch/ex16/codeb](#)

```
public class ComboBoxes extends JFrame {  
    private String[] description = {  
        "Ebullient", "Obtuse", "Recalcitrant", "Brilliant",  
        "Somnescient", "Timorous", "Florid", "Putrescent"  
    };  
    private JTextField textField = new JTextField(15);  
    private JComboBox comboBox = new JComboBox();  
    private JButton button = new JButton("Add items");  
    private int count = 0;  
  
    public ComboBoxes() {  
        for (int i = 0; i < 4; i++) {  
            comboBox.addItem(description[count++]);  
        }  
        textField.setEditable(true);  
        button.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                if (count < description.length) {  
                    comboBox.addItem(description[count++]);  
                }  
            }  
        });  
        comboBox.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                textField.setText("index: " + comboBox.getSelectedItem() + " " +  
                    ((JComboBox) e.getSource()).getSelectedItem());  
            }  
        });  
        setLayout(new FlowLayout());  
        add(textField);  
        add(comboBox);  
        add(button);  
        add(button);  
        add(button);  
        add(button);  
    }  
    public static void check() {  
        SwingConsole.run(new ComboBoxes(), 200, 175);  
    }  
}
```

## JList

- JList Список занимает на экране фиксированное число строк и не меняется
  - getSelectedvalues() выдает выбранные строки в виде объекта String
  - Ctrl+Select выбирает отдельные элементы
  - Shift+Select выбирает непрерывную группу элементов
- Пример. реализация JList lesson\_ch22/ch/ex16/codec
  - добавляет элементы в JList при помощи JButton
  - выбранные элементы JList автоматом отображаются в JTextArea
  - Component.new DefaultListModel() нужен именно для того, чтобы добавлять элементы в JList
- ВНИМАНИЕ. Если список фиксированный, можно сразу отдать ему массив значений без DefaultListModel
- ```
public class Lists extends JFrame {  
    private String[] flavors = {  
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",  
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",  
        "Praline Cream", "Mud Pie"  
    };  
    private DefaultListModel listItems = new DefaultListModel();  
    private JList list = new JList(listItems);  
    private JTextArea textArea = new JTextArea(flavors.length, 20);  
    private JButton button = new JButton("Add items");  
    private ActionListener bListener = new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            if (count < flavors.length) {  
                listItems.addElement(flavors[count++]);  
            } else {  
                button.setEnabled(false); // отключить кнопку  
            }  
        }  
    };  
    private ListSelectionListener listListener = new ListSelectionListener() {  
        @Override  
        public void valueChanged(ListSelectionEvent e) {  
            if (e.getValueIsAdjusting()) {  
                return;  
            }  
            textArea.setText("");  
            for (Object item : list.getSelectedValuesList()) {  
                textArea.append(item+"\n");  
            }  
        }  
    };  
    private int count = 0;  
    public Lists() {  
        textArea.setEditable(false);  
        setLayout(new FlowLayout());  
        Border border = BorderFactory.createMatteBorder(1, 1, 2, 2, Color.BLACK);  
        list.setBorder(border);  
        list.addListSelectionListener(listListener);  
        textArea.setBorderStyle(border);  
        button.addActionListener(bListener);  
        for (int i = 0; i < 4; i++) {  
            listItems.addElement(flavors[count++]);  
        }  
        add(textArea);  
        add(list);  
        add(button);  
    }  
    public static void check() {  
        SwingConsole.run(new Lists(), 250, 375);  
    }  
}
```

## JOptionPane

- JOptionPane Окна сообщений для информирования пользователя
    - это диалоговые окна разных типов Warning, Message, Input и так далее
  - Пример реализация сообщений на базе JOptionPane lesson\_ch22/ch/ex17/codeb
- ```
public class MessageBoxes extends JFrame {  
    private JButton[] buttons = {  
        new JButton("Alert"), new JButton("Yes/No"),  
        new JButton("Color"), new JButton("Input"),  
        new JButton("3 Vals")  
    };  
    private JTextField textField = new JTextField(15);  
    private ActionListener actionListener = new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            String id = ((JButton) e.getSource()).getText();  
  
            if (id.equals("Alert")) {  
                JOptionPane.showMessageDialog(null,  
                    "There's a bug on you!", "Hey!",  
                    JOptionPane.ERROR_MESSAGE);  
            } else if (id.equals("Yes/No")) {  
                JOptionPane.showConfirmDialog(null,  
                    "Yes or No", "choose yes",  
                    JOptionPane.YES_NO_OPTION);  
            } else if (id.equals("Color")) {  
                Object[] options = {"Red", "Green"};  
                int sel = JOptionPane.showOptionDialog(null,  
                    "Choose a Color!", "Warning",  
                    JOptionPane.DEFAULT_OPTION,  
                    JOptionPane.WARNING_MESSAGE,  
                    null, options, options[0]);  
                if (sel != JOptionPane.CLOSED_OPTION) { // если был сделан выбор  
                    textField.setText("Color selected:" + options[sel]);  
                }  
            } else if (id.equals("Input")) {  
                String val = JOptionPane.showInputDialog("How many lists do you see?");  
                textField.setText(val);  
            } else if (id.equals("3 Vals")) {  
                Object[] selections = {"First", "Second", "Third"};  
                Object val = JOptionPane.showInputDialog(null,  
                    "Choose one", "Input",  
                    JOptionPane.INFORMATION_MESSAGE,  
                    null, selections, selections[0]);  
                if (val != null) { // если был сделан выбор  
                    textField.setText(val.toString());  
                }  
            }  
        }  
    };  
    public MessageBoxes() {  
        setLayout(new FlowLayout());  
        for (int i = 0; i < buttons.length; i++) {  
            buttons[i].addActionListener(actionListener); // на все кнопки один Listener  
            add(buttons[i]);  
        }  
        add(textField);  
    }  
    public static void check() {  
        SwingConsole.run(new MessageBoxes(), 200, 200);  
    }  
}
```

## JMenu

- JMenu Меню объект меню в Swing
    - JMenuBar это объект куда можно набрать много JMenu в состав JMenuBar
    - JMenu собственно JMenu
    - JMenuItem это пункты меню, которые собирают в состав JMenu
    - JCheckBoxMenuItem вид пункта меню checkbox
  - Методы
    - setJMenuBar переключение JMenuBar при переходе на скажем другую форму
    - setMnemonic key shortcut быстрого доступа для ВИДИМЫХ пунктов меню или JButton
    - setAccelerator key shortcut быстрый доступ СКРЫТЫХ пунктов АКТИВНОГО JMenuBar
    - setActionCommand() назначает строку с именем команды, важно при интернационализации
    -
  - Пример. реализация JMenu на базе простого примера
- [lesson\\_ch22/ch/ex19/codea](#)
- ```
public class SimpleMenus extends JFrame {  
    private JTextField textField = new JTextField(15);  
    private ActionListener al = new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            JMenuBar jMenuBar = getJMenuBar();  
            textField.setText(((JMenuItem) e.getSource()).getText());  
        }  
    };  
  
    private JMenu[] menus = {  
        new JMenu("Winken"),  
        new JMenu("Blinken"),  
        new JMenu("Nod")  
    };  
    private JMenuItem[] items = {  
        new JMenuItem("Fee"), new JMenuItem("Fi"), new JMenuItem("Fo"),  
        new JMenuItem("Zip"), new JMenuItem("Zap"), new JMenuItem("Zot"),  
        new JMenuItem("Olly"), new JMenuItem("Oxen"), new JMenuItem("Free")  
    };  
  
    public SimpleMenus() {  
  
        for (int i = 0; i < items.length; i++) {  
            items[i].addActionListener(al);  
            menus[i % 3].add(items[i]); // раскидать по трем меню  
        }  
        JMenuBar jMenuBar = new JMenuBar();  
        for (JMenu menu : menus) {  
            jMenuBar.add(menu); // меню в менюбар  
        }  
        setJMenuBar(jMenuBar); // задать активный  
        setLayout(new FlowLayout());  
        add(textField);  
    }  
  
    public static void check() {  
        SwingConsole.run(new SimpleMenus(), 200, 150);  
    }  
}
```
-

## Каскадные меню

- Каскадные меню
- Пример. реализация JMenu на базе нескольких меню

lesson\_ch22/ch/ex19/codeb

```
public class Menus extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTextField textField = new JTextField("No flavor", 30);
    //JMenuBar1
    private JMenuBar jMenuBar1 = new JMenuBar();
    private JMenu jMenuF = new JMenu("File");
    private JMenu jMenuM = new JMenu("Flavors");
    private JMenu jMenuS = new JMenu("Safety");
    private JCheckBoxMenuItem[] safety = {
        new JCheckBoxMenuItem("Guard"),
        new JCheckBoxMenuItem("Hide")
    };
    private JMenuItem[] itemFiles = {new JMenuItem("Open")};
    private JMenuBar jMenuBar2 = new JMenuBar();
    private JMenu jMenuFoo = new JMenu("FooBar");
    private JMenuItem[] other = {
        new JMenuItem("Foo", KeyEvent.VK_F),
        new JMenuItem("Bar", KeyEvent.VK_A),
        new JMenuItem("Baz"),
    };
    JButton button = new JButton("Swap Menus");
    class BL implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            JMenuBar jMenuBar = getJMenuBar();
            setJMenuBar(jMenuBar == jMenuBar1 ? jMenuBar2 : jMenuBar1);
            validate();
        }
    }
    class ML implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            JMenuItem target = (JMenuItem) e.getSource();
            String actionCommand = target.getActionCommand();
            if (actionCommand.equals("Open")) {
                String s = textField.getText();
                boolean chosen = false;
                for (String flavor : flavors) {
                    if (s.equals(flavor)) {
                        chosen = true;
                    }
                }
                if (!chosen) {
                    textField.setText("Choose a flavor first!");
                } else {
                    textField.setText("Opening " + s + ". Mmm, mm!");
                }
            }
        }
    }
    class FL implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            JMenuItem target = (JMenuItem) e.getSource();
            textField.setText(target.getText());
        }
    }
}
```

- Пример(продолжение). реализация JMenu на базе нескольких меню

[lesson\\_ch22/ch/ex19/codeb](#)

```

• class FooL implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        textField.setText("Foo selected");
    }
}
class BarL implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        textField.setText("Bar selected");
    }
}
class BazL implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        textField.setText("Baz selected");
    }
}
class CMIL implements ItemListener {
    @Override
    public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target = (JCheckBoxMenuItem) e.getSource();
        String actionCommand = target.getActionCommand();
        if (actionCommand.equals("Guard")) {
            textField.setText("Guard the Ice Cream! " +
                "Guarding is " + target.getState()); // setState() может менять
        } else if (actionCommand.equals("Hide")) {
            textField.setText("Hide the Ice Cream! " +
                "Hiding is " + target.getState());
        }
    }
}
public Menus() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[1].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    int n = 0;
    for (String flavor : flavors) { // JMenu
        JMenuItem mItem = new JMenuItem(flavor);
        mItem.addActionListener(fl);
        jMenuM.add(mItem);
        if ((n++ + 1) % 3 == 0) {
            jMenuM.addSeparator();
        }
    }
    for (JCheckBoxMenuItem itemSafety : safety) { // JCheckBox
        jMenuS.add(itemSafety);
    }
    jMenuS.setMnemonic(KeyEvent.VK_A); // JMenuS
    jMenuF.add(jMenuS); // JMenuF
    jMenuF.setMnemonic(KeyEvent.VK_F);
    for (int i = 0; i < itemFiles.length; i++) {
        itemFiles[i].addActionListener(ml);
        jMenuF.add(itemFiles[i]); // добавить пункт меню.
    }
}

```

- Пример(продолжение). реализация JMenu на базе нескольких меню

[lesson\\_ch22/ch/ex19/codeb](#)

```
jMenuBar1.add(jMenuF); //jMenuBar1
jMenuBar1.add(jMenuM);
setJMenuBar(jMenuBar1);
textField.setEditable(false); //textField

add(textField, BorderLayout.CENTER);
button.addActionListener(new BL());
button.setMnemonic(KeyEvent.VK_S);
add(button, BorderLayout.NORTH);
for ( JMenuItem itemFBar : other) { //JMenuFoo
    jMenuFoo.add(itemFBar);
}
jMenuFoo.setMnemonic(KeyEvent.VK_B);
jMenuBar2.add(jMenuFoo); //jMenuBar2
}

public static void check() {
    SwingConsole.run(new Menus(), 300, 200);
}
}
```

## JPopupMenu

- JPopupMenu всплывающие меню

○ реализуются как внутренний класс обработчика сообщений MouseAdapter от MouseListener

- Пример. реализация JPopupMenu

[lesson\\_ch22/ch/ex21/codea](#)

```
public class Popup extends JFrame {
    private JPopupMenu jPopupMenu = new JPopupMenu();
    private JTextField textField = new JTextField(10);
    private void maybeShowPopup(MouseEvent e) {
        if (e.isPopupTrigger()) { // сработает по RMB
            jPopupMenu.show(e.getComponent(), e.getX(), e.getY());
        }
    }
    class PopupListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            maybeShowPopup(e);
        }
        public void mouseReleased(MouseEvent e) {
            maybeShowPopup(e);
        }
    }
    public Popup() {
        setLayout(new FlowLayout());
        add(textField);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                textField.setText(((JMenuItem) e.getSource()).getText());
            }
        };
        JMenuItem jMenuItem = new JMenuItem("Hinter");
        jMenuItem.addActionListener(al);
        jPopupMenu.add(jMenuItem);
        jMenuItem = new JMenuItem("Yon");
        jMenuItem.addActionListener(al);
        jPopupMenu.add(jMenuItem);
        PopupListener pl = new PopupListener();
        this.addMouseListener(pl); // сработать на всем поле JFrame
        textField.addMouseListener(pl); //сработать в пределах textField
    }
    public static void check() {
        SwingConsole.run(new Popup(), 300, 200 );
    }
}
```

-

## Рисование

- Рисование      использование Swing для создания графиков
  - для рисования используется JPanel и переопределенный метод paintComponent()
- Пример. реализация рисования синусоиды
  - JPanel панель вывода графика
  - JSlider слайдер перемещения по нарисованному графику
- ```
public class SineDraw extends JPanel {  
    private static final int SCALEFACTOR = 200;  
    private int cycles;  
    private int points;  
    private double[] sines;  
    private int[] pts;  
    public SineDraw() {  
        setCycles(5); // начальная установка  
    }  
    @Override  
    protected void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        int maxWidth = getWidth();  
        int maxHeight = getHeight();  
        double hStep = (double) maxWidth / (double) points;  
        pts = new int[points];  
        for (int i = 0; i < points; i++) {  
            pts[i] = (int) (sines[i] * maxHeight / 2 * 0.95 + maxHeight / 2);  
        }  
        g.setColor(Color.RED);  
        for (int i = 1; i < points; i++) {  
            int x1 = (int) ((i - 1) * hStep);  
            int x2 = (int) ((i) * hStep);  
            int y1 = pts[i - 1];  
            int y2 = pts[i];  
            g.drawLine(x1, y1, x2, y2);  
        }  
    }  
    public void setCycles(int cycles) { // меняет число периодов JSlider  
        this.cycles = cycles;  
        points = SCALEFACTOR * cycles * 2; // 2*PI*cycles  
        sines = new double[points];  
        for (int i = 0; i < points; i++) {  
            double radians = (Math.PI / SCALEFACTOR) * i;  
            sines[i] = Math.sin(radians);  
        }  
        repaint(); // перерисовать окно  
    }  
}  
• public class SineWave extends JFrame {  
    private SineDraw sines = new SineDraw();  
    private JSeparator adjustCycles = new JSeparator(1,30,5);  
    public SineWave() throws HeadlessException {  
        add(sines);  
        adjustCycles.addChangeListener(new ChangeListener() {  
            @Override  
            public void stateChanged(ChangeEvent e) { // отрабатывает JSlider  
                sines.setCycles(((JSeparator)e.getSource()).getValue());  
                System.out.println(((JSeparator)e.getSource()).getValue());  
            }  
        });  
        add(adjustCycles, BorderLayout.SOUTH);  
    }  
    public static void check() {  
        SwingConsole.run(new SineWave(), 700, 400);  
    }  
}
```

## JavaBean Components

- JavaBean Components
  - чтобы создать класс совместимый с JavaBeanComponent надо добавить Getters и Setters
- **ВНИМАНИЕ.** Чтобы класс был JavaBean Component надо добавить Getter и Setter на все независимые поля
- Пример. реализация официальная JavaBean Component [lesson\\_ch22/ch/ex21/exercise](#)

## JSlider рисование на панели

- JSlider рисование на панели [lesson\\_ch22/ch/ex24/exercise](#)

○ JSlider HORIZONTAL	ориентация задается
○ JSlider VERTICAL	ориентация задается
○ Graphics.drawOval()	прорисовка круга
○ Graphics.fillOval()	заполнение цветом
○ Graphics2D.setStroke()	толщина линии задается
○ Graphics2D.drawLine()	рисование линии задается
○ JPanel(new GridLayout)	размещение с использованием заполнения ячеек реализовано
○ List<Point> list	координаты запоминаются в списке и прорисовка идет по List<>
- Пример. реализация JSlider с рисованием объекта по List<Point> [lesson\\_ch22/ch/ex24/exercise](#)
- ```
public class SliderPoint extends JFrame {
    private final int SIZE_X = 480;
    private final int OFFSET_X = 80;
    private final int SIZE_Y = 350;
    private JSlider jsMoveY = new JSlider(1, SIZE_Y, SIZE_Y / 2);
    private JSlider jsMoveX = new JSlider(1, SIZE_X, SIZE_X / 2);
    private JTextField jtMoveY = new JTextField("Y:" + SIZE_Y / 2, 5);
    private JTextField jtMoveX = new JTextField("X:" + SIZE_X / 2, 5);
    private JButton jButton = new JButton("Clear Screen");
    private int radius = 10;
    private List<Point> list = new ArrayList<>();
    private boolean block = false;
    class CLY implements ChangeListener {
        @Override
        public void stateChanged(ChangeEvent e) { // отрабатывает JSlider
            int x = jsMoveX.getValue() + OFFSET_X;
            int y = SIZE_Y - jsMoveY.getValue();
            jtMoveY.setText(String.format("Y:%3d", y));
            if (block) { // если заблокирован, то не отрабатывать repaint
                return;
            }
            list.add(new Point(x, y));
            repaint();
        }
    }
    class CLX implements ChangeListener {
        public void stateChanged(ChangeEvent e) { // отрабатывает JSlider
            int x = jsMoveX.getValue() + OFFSET_X;
            int y = SIZE_Y - jsMoveY.getValue();
            jtMoveX.setText(String.format("X:%3d", x));
            if (block) { // если заблокирован, то не отрабатывать list, repaint
                return;
            }
            list.add(new Point(x, y));
            repaint();
        }
    }
    class AL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            initList(); // сбросить list
        }
    }
}
```

```

private JPanel jPanel = new JPanel() {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Point pStart = new Point(list.get(0));
        // начальная точка
        g.drawOval(pStart.x, pStart.y, radius, radius);
        g.setColor(Color.BLUE);
        g.fillOval(pStart.x, pStart.y, radius, radius);

        Graphics2D g2d = (Graphics2D) g;
        Stroke stroke = g2d.getStroke();
        g2d.setStroke(new BasicStroke(radius - 1));
        g2d.setColor(Color.BLUE);
        for (Point point : list) {
            g2d.drawLine(pStart.x + 5, pStart.y + 5, point.x + 5, point.y + 5);
            pStart = point;
        }
        // завершающую точку
        g2d.setStroke(stroke); // вернуть
        g.drawOval(pStart.x, pStart.y, radius, radius);
        g.setColor(Color.BLUE);
        g.fillOval(pStart.x, pStart.y, radius, radius);
    }
};

public SliderPoint() throws HeadlessException {
    jPanel.setPreferredSize(new Dimension(600, 400));
    jsMoveY.setPreferredSize(new Dimension(20, SIZE_Y));
    jsMoveY.addChangeListener(new CLY());
    jsMoveY.setOrientation(SwingConstants.VERTICAL);
    jtMoveY.setHorizontalAlignment(SwingConstants.CENTER);
    jtMoveY.setEditable(false);
    jsMoveX.setPreferredSize(new Dimension(SIZE_X, 20));
    jsMoveX.addChangeListener(new CLX());
    jsMoveX.setOrientation(SwingConstants.HORIZONTAL);
    jtMoveX.setHorizontalAlignment(SwingConstants.CENTER);
    jtMoveX.setEditable(false);
    jButton.addActionListener(new AL());
    JLabel jLabel = new JLabel("Color JPanel with Sliders");
    jLabel.setHorizontalAlignment(SwingConstants.CENTER);
    add(jLabel, BorderLayout.NORTH);
    add(jPanel);
    // add(jsSpeed);
    JPanel jpRed = new JPanel();
    jpRed.add(new JLabel("Y:"));
    jpRed.add(jsMoveY);
    jpRed.add(jtMoveY);
    add(jpRed, BorderLayout.EAST);
    // add(jsMoveX);
    JPanel jpMoveX = new JPanel();
    jpMoveX.setLayout(new BorderLayout(2, 3));
    jpMoveX.add(new JLabel("X:"), BorderLayout.WEST);
    jpMoveX.add(jsMoveX, BorderLayout.CENTER);
    jpMoveX.add(jtMoveX, BorderLayout.EAST);
    JPanel subPanel = new JPanel(new GridLayout(1, 5));
    subPanel.add(new JPanel()); // пропуск
    subPanel.add(new JPanel()); // пропуск
    subPanel.add(jButton); // кнопка по центру
    subPanel.add(new JPanel()); // пропуск
    subPanel.add(new JPanel()); // пропуск
    jpMoveX.add(subPanel, BorderLayout.SOUTH);
    add(jpMoveX, BorderLayout.SOUTH);
    initList(); // перерисовка автоматом
}
}

```

Пример. продолжение <>

[lesson\\_ch22/ch/ex24/exercise](#)

```
private void initList() {  
    list.clear();  
    list.add(new Point(SIZE_X / 2 + OFFSET_X, SIZE_Y / 2)); //две точки начало конец  
    block = true;  
    jsMoveX.setValue(SIZE_X / 2); // будет прорисовывать, надо заблокировать  
    jsMoveY.setValue(SIZE_Y / 2);  
    block = false;  
    repaint();  
}  
public static void check() {  
    SwingConsole.run(new SliderPoint(), 700, 450);  
}  
public static void main(String[] args) {  
    check();  
}  
}
```

## JPanel рисование на панели

- JPanel рисование на панели динамического графика
  - JPanel панель пользователя объединяет несколько полей
  - drawLine() рисование линий по массиву точек
- Пример. реализация динамического графика

[lesson\\_ch22/ch/ex27/exercise](#)

```
public class DrawCube extends JFrame {  
    private final int SIZE_X = 600;  
    private final int SIZE_Y = 300;  
  
    private final int OFFSET_X = 50; // 0 = 50 600 = 650  
    private final int OFFSET_Y = 300; // 0 = 300 300 = 0  
  
    private Point orgPoint = new Point(0, 0);  
    private Point yPoint = new Point(0, SIZE_Y); // шкала 300 10 на очко  
    private Point xPoint = new Point(SIZE_X, 0); // шкала 400 400/n на бросок  
    private int radius = 10; // радиус точки кривой  
    private Cube[] cubes = new Cube[5];  
    private JTextField[] jtMoves = new JTextField[6];  
    private JButton jButton = new JButton("Throw Cubes");  
    private List<Integer> list = new ArrayList<>(); // записать сумму бросков  
    class AL implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) { // отрабатывает JSlider  
            int sum = 0;  
            for (int i = 0; i < cubes.length; i++) {  
                sum += cubes[i].getValue();  
                jtMoves[i].setText(String.format("Cube%d: %d", i, cubes[i].getLast()));  
            }  
            jtMoves[5].setText(String.format("Sum: %2d", sum));  
            list.add(sum);  
            repaint();  
        }  
    }  
    private void initList() {  
        list.clear();  
        repaint();  
    }  
    private int CX(int x) {  
        return x + OFFSET_X;  
    }  
    private int CY(int y) {  
        return OFFSET_Y - y;  
    }
```

- Пример.продолжение<>

[lesson\\_ch22/ch/ex27/exercise](#)

```

• private JPanel jPanel = new JPanel() {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        // оси
        g.drawLine(cX(orgPoint.x), cY(orgPoint.y), cX(orgPoint.x), cY(yPoint.y));
        g.drawLine(cX(orgPoint.x), cY(orgPoint.y), cX(xPoint.x), cY(orgPoint.y));

        for (int i = 5; i <= 30; i += 5) { // насечки на y
            g.drawLine(cX(orgPoint.x - 5), cY(orgPoint.y + i * 10),
                       cX(orgPoint.x + 5), cY(orgPoint.y + i * 10));
        }
        if (list.size() == 0) {
            return; // нет значений
        }
        int count = list.size(); // число точек

        int stepX = SIZE_X; // число точек на интервале начинаем с нуля
        if (list.size() > 1) {
            stepX = SIZE_X / (count - 1); // число точек на интервале с нуля
        }
        for (int i = stepX; i <= SIZE_X; i += stepX) {
            g.drawLine(cX(i), cY(orgPoint.y - 5), cX(i), cY(orgPoint.y + 5));
        }
        // значения
        int stepY = 10;
        int x = orgPoint.x; // начальная координата
        int y = orgPoint.y;
        int xOld = x; // начальная координата
        int yOld = y;
        Graphics2D g2d = (Graphics2D) g;
        Stroke stroke = g2d.getStroke();
        g2d.setColor(Color.BLUE);

        for (int i = 0; i < list.size(); i++) {
            // начальная точка
            y = list.get(i) * stepY; // новое значение
            if (i > 0) {
                x = xOld + stepX; // новое значение
                g2d.setStroke(new BasicStroke(radius - 3));
                g2d.drawLine(cX(xOld), cY(yOld), cX(x), cY(y));
            }
            g2d.setStroke(stroke);
            g2d.drawOval(cX(x - radius / 2), cY(y + radius / 2), radius, radius); //
            на новом месте
            g2d.fillOval(cX(x - radius / 2), cY(y + radius / 2), radius, radius);
            xOld = x;
            yOld = y;
        }
    }
};

```

- Пример. продолжение<>

[lesson\\_ch22/ch/ex27/exercise](#)

```

public DrawCube() throws HeadlessException {
    setLayout(new BorderLayout());
    for (int i = 0; i < cubes.length; i++) {
        cubes[i] = new Cube();
    }
    jPanel.setPreferredSize(new Dimension(600, 400));
    jButton.addActionListener(new AL());
    JLabel jLabel = new JLabel("Cube Game");
    jLabel.setHorizontalAlignment(SwingConstants.CENTER);
    add(jLabel, BorderLayout.NORTH);
    add(jPanel);
    JPanel jpMove = new JPanel(new GridLayout(2, 6)); // первая строка
    for (int i = 0; i < jtMoves.length - 1; i++) {
        jtMoves[i] = new JTextField("Cube" + i + ": ");
        jtMoves[i].setEditable(false);
        jpMove.add(jtMoves[i]);
    }
    jtMoves[5] = new JTextField("Sum: 0");
    jtMoves[5].setEditable(false);
    jpMove.add(jtMoves[5]); // вторая строка
    jpMove.add(new JPanel()); // пропуск
    jpMove.add(new JPanel()); // пропуск
    jpMove.add(jButton); // кнопка по центру
    add(jpMove, BorderLayout.SOUTH);
}
public static void check() {
    SwingConsole.run(new DrawCube(), 700, 450);
}

```

## JDialog диалоговые окна

- JDialog диалоговые окна, временные окна для выполнения операций, работают также как JFrame
  - setVisible() показывает окно JDialog,
  - dispose() скрывает окно JDialog
- Пример. реализация простого JDialog

[lesson\\_ch22/ch/ex29/codea](#)

```

public class MyDialog extends JDialog {
    public MyDialog(Frame owner) {
        super(owner);
        setLayout(new FlowLayout());
        add(new JLabel("Here is my dialog"));
        JButton jBOK = new JButton("OK");
        jBOK.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose(); // закрыть диалог
            }
        });
        add(jBOK);
        setSize(150, 125); // размер диалога
    }
}
public class Dialogs extends JFrame {
    private JButton jButton = new JButton("Dialog Box");
    private MyDialog myDialog = new MyDialog(null); // null это parent JFrame
    public Dialogs() throws HeadlessException {
        jButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                myDialog.setVisible(true); // открыть диалог
            }
        });
        add(jButton);
    }
    public static void check() {
        SwingConsole.run(new Dialogs(), 125, 75);
    }
}

```

- Пример. реализация JDialog с GridLayout игра TicTacToe «крестики нолики» [lesson\\_ch22/ch/ex29/codeb](#)

## JFileChooser

- JFileChooser диалоговое окно выбора файлов
  - это стандартное окно выбора файлов
- JColorChooser диалоговое окно выбора цветов
  - это стандартное окно выбора цветов
- Пример. реализация JFileChooser
- Пример. реализация JColorChooser

```
lesson_ch22/ch/ex29/codec  
lesson_ch22/ch/ex29/exercise

public class JFileChooserTest extends JFrame {  
    private JTextField jtFileName = new JTextField();  
    private JTextField jtDir = new JTextField();  
    private JButton jbOpen = new JButton("Open");  
    private JButton jbSave = new JButton("Save");  
  
    class OpenL implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            JFileChooser jFileChooser = new JFileChooser();  
            int rVal = jFileChooser.showOpenDialog(JFileChooserTest.this); // класс ?  
            if (rVal == JFileChooser.APPROVE_OPTION) {  
                jtFileName.setText(jFileChooser.getSelectedFile().getName());  
                jtDir.setText(jFileChooser.getCurrentDirectory().toString());  
            }  
            if (rVal == JFileChooser.CANCEL_OPTION) {  
                jtFileName.setText("You pressed cancel");  
                jtDir.setText("");  
            }  
        }  
    }  
    class SaveL implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            JFileChooser jFileChooser = new JFileChooser();  
            int rVal = jFileChooser.showOpenDialog(JFileChooserTest.this); // класс ?  
            if (rVal == JFileChooser.APPROVE_OPTION) {  
                jtFileName.setText(jFileChooser.getSelectedFile().getName());  
                jtDir.setText(jFileChooser.getCurrentDirectory().toString());  
            }  
            if (rVal == JFileChooser.CANCEL_OPTION) {  
                jtFileName.setText("You pressed cancel");  
                jtDir.setText("");  
            }  
        }  
    }  
    public JFileChooserTest() throws HeadlessException {  
        JPanel jPanel = new JPanel();  
        jbOpen.addActionListener(new OpenL());  
        jPanel.add(jbOpen);  
        jbSave.addActionListener(new SaveL());  
        jPanel.add(jbSave);  
        add(jPanel, BorderLayout.SOUTH);  
        jtDir.setEditable(false);  
        jtFileName.setEditable(false);  
        jPanel = new JPanel();  
        jPanel.setLayout(new GridLayout(2, 1));  
        jPanel.add(jtFileName);  
        jPanel.add(jtDir);  
        add(jPanel, BorderLayout.NORTH);  
    }  
    public static void check() {  
        SwingConsole.run(new JFileChooserTest(), 250, 150);  
    }  
}
```

## HTML for Components

- HTML for Component это возможность для компонентов Swing отображать HTML текст
  - HTML работает с JButton, JTabbedPane, JMenuItem, JToolTip, JRadioButton, JCheckBox
- Пример. реализация HTML для компонентов Swing
- **ВНИМАНИЕ.** Реализовано динамическое ДОБАВЛЕНИЕ компонента по нажатию кнопки
- ```
public class HTMLButton extends JFrame {  
    private JButton jButton = new JButton("<html><b><font size = +2>" +  
        "<center>Hello!<br><i>Press me now!"  
    );  
    public HTMLButton() throws HeadlessException {  
        jButton.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                add(new JLabel("<html><i><font size = +4>WORLD!"));  
                validate(); // перерисовать  
            }  
        });  
        setLayout(new FlowLayout());  
        add(jButton);  
    }  
    public static void check() {  
        SwingConsole.run(new HTMLButton(), 200, 500);  
    }  
}
```

[lesson\\_ch22/ch/ex30/codea](#)

## HTML Components and JToolTip

- HTML Components and JToolTip
  - JToolTip поддержка для JButton и JMenuItem
  - HTML поддержка JButton, JTabbedPane, JMenuItem, JToolTip, JRadioButton, JCheckBox
- Пример. реализация HTML для всех поддерживающих его компонентов
- 
- Пример. продолжение<>

[lesson\\_ch22/ch/ex30/exercise](#)

```
public class HTMLComponents extends JFrame {  
    String s = "<html><b><font size = +2<center>Hello!<br><i>Press me now!";  
    private JButton jButton = new JButton("<html><center><i>JButton!");  
    private JButton jButton2 = new JButton("<html><b><center>Hello2!");  
    private JMenuItem[] jMenus = new JMenuItem[5];  
    private JTabbedPane jTabbedPane = new JTabbedPane(JTabbedPane.TOP);  
    private JToolTip jToolTipC = new JToolTip();  
    private JRadioButton[] jRadioButtons = new JRadioButton[5];  
    private JCheckBox[] jCheckboxes = new JCheckBox[5];  
  
    private JButton jButtonT = new JButton("<html><center><i>JButtonT!");  
    @Override  
    public JToolTip createToolTip() {  
        JToolTip jToolTip = new JToolTip()  
        @Override  
        public String getTipText() {  
            String s = ("<html><b><center> Description for: <i>JButton");  
            return s;  
        }  
    };  
    JOptionPane.showMessageDialog(null, "ToolTip processed",  
        "Title", JOptionPane.INFORMATION_MESSAGE);  
    return jToolTip;  
};
```

- Пример. продолжение<>

[lesson\\_ch22/ch/ex30/exercise](#)

```

private JButton jButtonT2 = new JButton("<html><center><i>JButtonT2!" ) {
    @Override
    public JToolTip createToolTip() {
        JToolTip jToolTip = new JToolTip() {
            @Override
            public String getTipText() {
                String s = ("<html><b><center> Description for: <i>JButtonT2");
                return s;
            }
        };
        return jToolTip;
    }
};

// JRadioButton, JCheckBox
private class AL implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Button pressed",
                                   "Title", JOptionPane.INFORMATION_MESSAGE);
    }
}

public HTMLComponents() throws HeadlessException {
    //menus
    for (int i = 0; i < jMenus.length; i++) {
        jMenus[i] = new JMenu("<html><b><center>Menu:" + i);
        for (int j = 0; j < 5; j++) {
            JMenuItem jMenuItem =
                new JMenuItem("<html><b><center>Menu:" + i + "<i> item:" + i + j) {
                    @Override
                    public JToolTip createToolTip() {
                        JToolTip jToolTip = new JToolTip() {
                            @Override
                            public String getTipText() {
                                String s = ("<html><b><center> Description for: item"));
                                return s;
                            }
                        };
                        return jToolTip;
                    }
                };
            jMenuItem.setToolTipText("");
            jMenus[i].add(jMenuItem);
        }
    }
}

//JButtons
jButton.addActionListener(new AL());
jButton.setToolTipText("<html><b><center> Description for: <i>JButton");
jButton2.addActionListener(new AL());
jButton2.setToolTipText("<html><b><center> Description for: <i>JButton2");
jButtonT.setToolTipText(""); // отрабатывает внутренняя функция
jButtonT2.setToolTipText(""); // отрабатывает внутренняя функция
setLayout(new FlowLayout());
JMenuBar jMenuBar = new JMenuBar();
jMenuBar.setName("MTab");
for (JMenu jMenu : jMenus) {
    jMenuBar.add(jMenu);
}
JPanel jPanel = new JPanel();
jPanel.setName("BTab");
jPanel.add(jButton);
jPanel.add(jButton2);
jPanel.add(jButtonT);
jPanel.add(jButtonT2);

```

- Пример. продолжение<>

[lesson\\_ch22/ch/ex30/exercise](#)

```

        JPanel jPanel2 = new JPanel(); // radiobuttons
        jPanel2.setName("RTab");
        ButtonGroup buttonGroup = new ButtonGroup();
        for (int i = 0; i < jRadioButtons.length; i++) {
            jRadioButtons[i] = new JRadioButton("<html><b><center>RB:" + i);
            buttonGroup.add(jRadioButtons[i]);
            jPanel2.add(jRadioButtons[i]);
        }
        jRadioButtons[0].setSelected(true);
        Random rnd = new Random(); // checkbox
        JPanel jPanel3 = new JPanel();
        jPanel3.setName("CTab");
        ButtonGroup buttonGroup2 = new ButtonGroup();
        for (int i = 0; i < jCheckBoxes.length; i++) {
            jCheckBoxes[i] = new JCheckBox("<html><b><center>CB:" + i);
            buttonGroup2.add(jCheckBoxes[i]);
            jPanel3.add(jCheckBoxes[i]);
        }
        jCheckBoxes[0].setSelected(true);
        jTabbedPane.add(jMenuBar);
        jTabbedPane.add(jPanel);
        jTabbedPane.add(jPanel2);
        jTabbedPane.add(jPanel3);
        add(jTabbedPane);
    }
    public static void check() {
        SwingConsole.run(new HTMLComponents(), 400, 200);
    }
}

```

## JProgressBar

- JProgressBar              индикатор работает обычно вместе с JSlider
- ProgressMonitor          индикатор с расширенным функционалом
- Пример. реализация JProgressBar и ProgressMonitor

[lesson\\_ch22/ch/ex31/codea](#)

```

public class Progress extends JFrame {
    private JProgressBar jProgressBar = new JProgressBar();
    private ProgressMonitor progressMonitor = new ProgressMonitor(
        this, "Monitoring Progress", "Test", 0, 100 );
    private JSlider jSlider = new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public Progress() throws HeadlessException {
        setLayout(new GridLayout(2,1));
        add(jProgressBar);
        progressMonitor.setProgress(0);
        progressMonitor.setMillisToPopup(1000); // 1 sec
        jSlider.setValue(0);
        jSlider.setPaintTicks(true);
        jSlider.setMajorTickSpacing(20);
        jSlider.setMinorTickSpacing(5);
        jSlider.setBorder(new TitledBorder("Slide Me"));
        jProgressBar.setModel(jSlider.getModel()); // модель общую JProgrssBar и JSlider
        add(jSlider);
        jSlider.addChangeListener(new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                progressMonitor.setProgress(jSlider.getValue()); // ProgressMonitor
            }
        });
    }
    public static void check() {
        SwingConsole.run(new Progress(), 300, 200);
    }
}

```

## Pluggable Look and Feel

- Pluggable Look and Feel      модульный интерфейс пользователя
  - Metal      платформенно независимый интерфейс      «cross»
  - System      стандартный интерфейс      «system»
  - Custom      интерфейс из библиотеки Sun      «motif»
- Пример. реализация интерфейса программы      [lesson\\_ch22/ch/ex33/codea](#)
- ```
public class LookAndFeel extends JFrame {
    private String[] choices = "Eeny Meeny Minnie Mickey Moe Larry Curly".split(" ");
    private Component[] samples = {
        new JButton("JButton"),
        new JTextField("JTextField"),
        new JLabel("JLabel"),
        new JCheckBox("JCheckBox"),
        new JRadioButton("JRadioButton"),
        new JComboBox(choices),
        new JList(choices)
    };
    public LookAndFeel() throws HeadlessException {
        super("Look And Feel");
        setLayout(new FlowLayout());
        for (Component component : samples) {
            add(component);
        }
    }
    private static void usageError() {
        System.out.println("Usage: LookAndFeel [cross|system|motiff]");
        //System.exit(1);
    }
    public static void check(String[] args) {
        Random rnd = new Random();
        if (args.length == 0) {
            usageError();
        }
        // args = new String[]{"cross"};
        // args = new String [] {"system"};
        args = new String [] {"motif"};
    }
    if (args[0].equals("cross")) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else if (args[0].equals("system")) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else if (args[0].equals("motif")) {
        try {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else {
        usageError();
    }
    SwingConsole.run(new LookAndFeel(), 300, 300);
}
```

## JNLP Java Network Launch Protocol

- JNLP Java Network Launch Protocol протокол запуска приложений по сети
  - позволяет загрузить приложение по сети на машину клиента
  - позволяет запустить приложение с сайта, с которого было загружено
  - позволяет приложению динамически получать данные из интернета
  - позволяют приложению проверять версию из интернета
- Security JNLP
  - архивы jar с приложениями должны иметь цифровую подпись
  - архивы jar без подписи в любом случае получают доступ к некоторым ресурсам системы клиента
- Реализация JNLP
  - создается стандартное приложение JAVA
  - упаковывается в jar архив
  - создается XML файл запуска архива jar
  - архив jar подписывается цифровой подписью
  - если подписи нет, приложение может запросить доступ у клиента напрямую

## Реализация JNLP на базе JAR, XML, HTML

- Компиляция можно сделать как вручную так и на базе IDEA
- Компиляция вручную
  - делать в любой папке добавится только путь ch22.ex33.codeb.CodeB
  - подключить –classpath JDK\_HOME\javaws.jar и указать путь до других файлов исходников
  - подключить -d <path> указать папку куда скомпилировать файлы
  - **ВНИМАНИЕ.** архив JAR делать из папки -d <path>, файлы \*class доставать из пути package
- Создание архива JAR
  - строка компиляции архива jar cvf jnlpfilechooser.jar <package\_path>\\*.class
  - <package\_path> путь до файлов, который в папке -d<path> создаст компилятор
- Настройка XML файла
  - скопировать файлы html,xml(jnlp),jar в какую нибудь папку, например D:\temp\jar
  - XML прописать путь <coodebase> D:\temp\jar
  - XML вписать название файла jnlp, jar
  - XML указать класс с main() если есть package указать путь через точку package.path.to.Class
- Настройка HTML файла задать имя XML (jnlp) файла

```
html>
Follow the instructions in JnlpFileChooser.java to
build jnlpfilechooser.jar, then download and open filechooser:
<a href="filechooser.jnlp">click here</a>
</html>
```

- Пример. реализация JNLP Модуль компиляции запускает cmd файл

lesson\_ch22/ch/ex33/codeb

```
public class JMakeR {
    public static void main(String[] args) {
        try {
            Process p = new ProcessBuilder("cmd /c .\\src\\ch22\\ex33\\codeb\\jarpack\\
                jcmd2.cmd ".split(" ")).inheritIO().start();
            Thread.sleep(1000);
            p = new ProcessBuilder("cmd /c .\\src\\ch22\\ex33\\codeb\\jarpack\\
                jcmd22.cmd ".split(" ")).inheritIO().start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- Пример. реализация JNLP Основной модуль

[lesson\\_ch22/ch/ex33/codeb](#)

```

• public class JnlpFileChooser extends JFrame {
    private JTextField jtFileName = new JTextField();
    private JButton jbOpen = new JButton("Open");
    private JButton jbSave = new JButton("Save");
    private JEditorPane jEditorPane = new JEditorPane();
    private JScrollPane jScrollPane = new JScrollPane();
    private FileContents fileContents;
    class OpenL implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            FileOpenService fileOpenService = null;
            try {
                fileOpenService = (FileOpenService) ServiceManager.lookup(
                    "javax.jnlp.FileOpenService");
            } catch (UnavailableServiceException ex) {
                throw new RuntimeException(ex);
            }
            if (fileOpenService != null) {
                try {
                    fileContents = fileOpenService.openFileDialog(
                        ".", new String[]{"txt", "*"});
                    if (fileContents == null) {
                        return;
                    }
                    jtFileName.setText(fileContents.getName());
                    jEditorPane.read(fileContents.getInputStream(), null);
                } catch (IOException ex) {
                    throw new RuntimeException(ex);
                }
                jbSave.setEnabled(true); // activate button
            }
        }
    }

    class SaveL implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            FileSaveService fileSaveService = null;
            try {
                fileSaveService = (FileSaveService) ServiceManager.lookup(
                    "javax.jnlp.FileSaveService");
            } catch (UnavailableServiceException ex) {
                throw new RuntimeException(ex);
            }
            if (fileSaveService != null) {
                try {
                    fileContents = fileSaveService.saveFileDialog(
                        ".", new String[]{"txt"}, 
                        new ByteArrayInputStream(jEditorPane.getText().getBytes()),
                        fileContents.getName());
                    if (fileContents == null) {
                        return;
                    }
                    jtFileName.setText(fileContents.getName());
                } catch (IOException ex) {
                    throw new RuntimeException(ex);
                }
                jbSave.setEnabled(true); // activate button
            }
        }
    }
}

```

- Пример. продолжение <> реализация JNLP

[lesson\\_ch22/ch/ex33/codeb](#)

```

• public JnlpFileChooser() throws HeadlessException {
    JPanel jPanel = new JPanel();
    jbOpen.addActionListener(new OpenL());
    jbSave.addActionListener(new SaveL());
    jbSave.setEnabled(false);
    jPanel.add(jbOpen);
    jPanel.add(jbSave);
    jEditorPane.setContentType("text"); // panel content
    jScrollPane.setViewportView(jPanel);
    add(jScrollPane, BorderLayout.CENTER);
    add(jPanel, BorderLayout.SOUTH);
    jtFileName.setEditable(false);
    JLabel jLabel = new JLabel("<html>JNLP File Chooser",
    jPanel = new JPanel();
    jPanel.setLayout(new GridLayout(2,1));
    jPanel.add(jtFileName);
    jPanel.add(jLabel);
    add(jPanel, BorderLayout.NORTH);
}
public static void check() {
    JnlpFileChooser jnlpFileChooser = new JnlpFileChooser();
    jnlpFileChooser.setSize(400,300);
    jnlpFileChooser.setVisible(true);
}
}

• public class CodeB {
    public static void main(String[] args) {
        JnlpFileChooser.check();
    }
}

```

- Пример. Реализация JNLP Файл cmd компиляции, запуск из >>jb01\

[lesson\\_ch22/ch/ex33/codeb](#)

```

• javac -cp "C:\Program Files\Java\jdk1.8.0_112\jre\lib\javaws.jar;.\src" -d
  .\src\ch22\ex33\codeb\jarpack .\src\ch22\ex33\codeb\CodeB.java
  cd src\ch22\ex33\codeb\jarpack
  jar cvf jnlpfilechooser.jar ch22\ex33\codeb/*.class
  cd ..\..\..\..\..
  mkdir D:\temp2\_jarpack
  copy src\ch22\ex33\codeb\jarpack\* D:\temp2\_jarpack

```

- Пример. Реализация JNLP Файл XML

[lesson\\_ch22/ch/ex33/codeb](#)

```

<?xml version="1.0" encoding="UTF-8" ?>
<jnlp spec = "1.0+"
  codebase="file:///D:/temp2/_jarpack"
  href="filechooser.jnlp">
  <information>
    <title>FileChooser demo application</title>
    <vendor>Mindview Inc.</vendor>
    <description>
      Jnlp File chooser Application
    </description>
    <description kind="short">
      Demonstrates opening, reading and writing a text file
    </description>
    <icon href="mindview.gif"/>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.3+"
      href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="jnlpfilechooser.jar" download="eager"/>
  </resources>
  <application-desc
    main-class="ch22.ex33.codeb.CodeB"/>
</jnlp>

```

## Параллельное выполнение Swing

- Параллельное выполнение Swing
  - все потоки в Swing запускаются диспетчером SwingUtilities.invokeLater()
- Продолжительная задача и Поток Диспетчеризации
- **ВНИМАНИЕ.** НЕ ИСПОЛЬЗОВАТЬ поток диспетчеризации (поток main() ) для Продолжительной Задачи
- Пример. реализация пуска задач из потока диспетчеризации (main()короче) [lesson\\_ch22/ch/ex33/coded](#)
  - Недостатки
  - во первых      остаются висеть не завершенные задачи в очереди
  - во вторых      при повторном нажатии кнопки запуска задачи вылетает Exception
- ```
public class InterruptableLongRunningTask extends JFrame {  
    private JButton jButton = new JButton("Start Long Running Task");  
    private JButton jButton2 = new JButton("End Long Running Task");  
    ExecutorService exec = Executors.newSingleThreadExecutor(); // один поток  
  
    public InterruptableLongRunningTask() {  
        jButton.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                //  
                //  
                //  
                //  
                if (exec.isShutdown()) {  
                    System.out.println("ExecutorService is shutdown");  
                    return;  
                }  
                Task task = new Task();  
                exec.execute(task);  
                System.out.println(task+" added to the queue");  
            }  
        });  
        jButton2.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) { // запуск прерывания  
                exec.shutdownNow(); // вырубить все задачи  
  
            }  
        });  
        setLayout(new FlowLayout());  
        add(jButton);  
        add(jButton2);  
    }  
    public static void check() {  
        SwingConsole.run(new InterruptableLongRunningTask(), 200, 150);  
    }  
}
```

## Работа с объектами Callable в Swing

- Корректное завершение программы
  - все потоки набираются в List<Future>
  - при завершении программы список отрабатывается по элементам
- Пример. реализация запуска потоков для Swing с объектами Future [lesson\\_ch22/ch/ex33/codee](#)
  - Task            элементарная задача
  - CallableTask    расширение до Callable<String> чтобы использовать Future<String> механизм
  - TaskItem        элемент списка, состоит из объекта Future<String> и задачи Callable<String>
  - TaskManager     менеджер задач, запускает, останавливает, выводит результаты
- Пример. реализация объекта Task [lesson\\_ch22/ch/ex33/codee](#)
- ```
public class Task implements Runnable {
    private static int counter = 0;
    private final int id = counter++;

    @Override
    public void run() {
        System.out.println(this + " started");
        try {
            TimeUnit.MILLISECONDS.sleep(3000);
        } catch (InterruptedException e) {
            System.out.println(this + " interrupted");
            return;
        }
        System.out.println(this + " completed");
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "Task " + id;
    }
}
```
- Пример. реализация объекта TaskItem<R,C extends Callable<R>> [lesson\\_ch22/ch/ex33/codee](#)

```
public class TaskItem<R,C extends Callable<R>> {
    public final Future<R> future; // R возвращаемое значение, Future это же объект Callable то есть должен возвращать
    public final C task; // а это просто Callable<R> можно было не задавать, если только дальше нет потомков Callable<R>

    public TaskItem(Future<R> future, C task) {
        this.future = future;
        this.task = task;
    }
}
```
- Пример. реализация объекта CallableTask<String> [lesson\\_ch22/ch/ex33/codee](#)

```
public class CallableTask extends Task implements Callable<String> { // берем нашу задачу и превращаем в Callable
    @Override
    public String call() throws Exception { // поэтому и список <C extends Callable<R>>
        run(); //запихнут CallableTask == Callable<String>
        return "Return value of "+this; // от предка sleep(3sec)
    }
}
```

- Пример. реализация объекта TaskManager<R, C extends Callable<R>>

[lesson\\_ch22/ch/ex33/codee](#)

```

public class TaskManager<R, C extends Callable<R>> extends ArrayList<TaskItem<R, C>> {
    private ExecutorService exec = Executors.newSingleThreadExecutor(); // одна задача
    public void add(C task) {   // добавить Callable<String> задачу
        add(new TaskItem<R, C>(exec.submit(task), task)); // <Future, Callable task>
    }   // тут же запустить ее через submit
    public List<R> getResults() {
        Iterator<TaskItem<R, C>> items = iterator(); // прогнать объект по элементам
        List<R> results = new ArrayList<R>();
        while (items.hasNext()) {
            TaskItem<R, C> item = items.next();
            if (item.future.isDone()) { // проверяем объект future задачи
                try {
                    results.add(item.future.get()); // получить результат

                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
                items.remove();
            }
        }
        return results;
    }

    public List<String> purge() {
        Iterator<TaskItem<R, C>> items = iterator();
        List<String> results = new ArrayList<String>();
        while (items.hasNext()) {
            TaskItem<R, C> item = items.next();
            if (!item.future.isDone()) { // проверяем объект future задачи
                results.add("Cancelling " + item.task); // выдать название задачи
                item.future.cancel(true); // отключение все таки через Future<>
                items.remove();
            }
        }
        return results;
    }

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        String s = "abcdefghijklmnopqrstuvwxyz";
        for (int i = 0; i < s.length(); i++) {
            list.add(s.substring(i, i + 1));
        }
        List<String> list2 = new ArrayList<>(list);
        Iterator<String> it = list.iterator();
        int count = 0;
        while (it.hasNext()) { // так работает
            String sNext = it.next();
            System.out.println(sNext);
            if (count++ % 3 == 0) {
                it.remove();
            }
        }
        System.out.println(list);
        count = 0;
        for (String s2 : list2) { // так не работает
            System.out.println(s2);
            if (count++ % 3 == 0) {
                list2.remove(s2);
            }
        }
    }
}

```

- Пример. реализация объекта InterruptableLongRunningCallable [lesson\\_ch22/ch/ex33/codee](#)
- ```

public class InterruptableLongRunningCallable extends JFrame {
    private JButton jButton = new JButton("Start Long Running Task");
    private JButton jButton2 = new JButton("End Long Running Task");
    private JButton jButton3 = new JButton("Get results");
    private TaskManager<String, CallableTask> manager = new TaskManager<>();
    public InterruptableLongRunningCallable() { //используется Callable<String>
        jButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                CallableTask task = new CallableTask();
                manager.add(task); // и тут же запустили
                System.out.println(task+" added to the queue");
            }
        });
        jButton2.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // List<String> list = manager.purge();
                for (String result : manager.purge()) { // отрубить все незавершенные
                    System.out.println(result); // задачи и вернуть список
                }
            }
        });
        jButton3.addActionListener(new ActionListener() { // собираем честные результаты
            @Override
            public void actionPerformed(ActionEvent e) {
                for (TaskItem<String, CallableTask> taskItem : manager) {
                    taskItem.task.getId();
                }
                List<String> list = manager.getResults();
                for (String result : manager.getResults()) { // найти завершенные задачи
                    System.out.println(result); // и вернуть список
                }
            }
        });
        setLayout(new FlowLayout());
        add(jButton);
        add(jButton2);
        add(jButton3);
    }
    public static void check() {
        SwingConsole.run(new InterruptableLongRunningCallable(), 200, 150);
    }
}

```

## Swing Callable Task and Progress Monitor

- Использование Progress Monitor для визуальной поддержки задач [lesson\\_ch22/ch/ex33/codef](#)
- все тоже самое, но в задаче запускается ProgressMonitor
- если монитор закрывается пользователем, задача прерывает сама себя
- Пример. реализация Callable<String> и ProgressMonitor [lesson\\_ch22/ch/ex33/codef](#)

```

public class MonitoredCallable implements Callable<String> {
    private static int counter = 0;
    private final int id = counter++;
    private final ProgressMonitor monitor;
    private final static int MAX = 8;

    public MonitoredCallable(ProgressMonitor monitor) {
        this.monitor = monitor;
        monitor.setNote(toString()); // метод монитора Note пишем свой this.toString()
        monitor.setMaximum(MAX - 1);
        monitor.setMillisToPopup(500); // время до всплытия
    }
}

```

- Пример. продолжение<> Callable<String>

[lesson\\_ch22/ch/ex33/codef](#)

```

public String call() throws Exception {
    System.out.println(this + " started");
    try {
        for (int i = 0; i < MAX; i++) {
            TimeUnit.MILLISECONDS.sleep(500);
            if (monitor.isCanceled()) { // throw new InterruptException() работает
                Thread.currentThread().interrupt(); // return работает
            }
            final int progress = i;
            SwingUtilities.invokeLater(new Runnable() { // задание Swing monitor
                public void run() {
                    monitor.setProgress(progress);
                }
            });
        }
    } catch (InterruptedException e) {
        monitor.close();
        System.out.println(this + " interrupted");
        return "Result: " + this + " interrupted";
    }
    System.out.println(this + " completed");
    return "Result: " + this + " completed";
}
public String toString() { return "Task " + id;
}
}

```

- Пример. реализация Callable<String> и ProgressMonitor

[lesson\\_ch22/ch/ex33/codef](#)

```

public class MonitoredLongRunningCallable extends JFrame {
    private JButton jButton = new JButton("Start Long Running Task");
    private JButton jButton2 = new JButton("End Long Running Task");
    private JButton jButton3 = new JButton("Get results");
    private TaskManager<String, MonitoredCallable> manager = new TaskManager<>();
    public MonitoredLongRunningCallable() {
        jButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MonitoredCallable task = new MonitoredCallable(
                    new ProgressMonitor( MonitoredLongRunningCallable.this,
                        "Long Running Task", "", 0, 0));
                manager.add(task); // и тут же запустили
                System.out.println(task + " added to the queue");
            }
        });
        jButton2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for (String result : manager.purge()) { // отрубить незавершенные задачи
                    System.out.println(result);
                }
            }
        });
        jButton3.addActionListener(new ActionListener() { // собираем результаты
            public void actionPerformed(ActionEvent e) {
                for (String result : manager.getResults()) { // найти завершенные задачи
                    System.out.println(result);
                }
            }
        });
        setLayout(new FlowLayout());
        add(jButton);
        add(jButton2);
        add(jButton3);
    }
    public static void check() {
        SwingConsole.run(new MonitoredLongRunningCallable(), 200, 500);
    }
}

```

## Визуальные потоки

- Визуальные потоки
  - добавление задач в объекты Swing
  - запуск локальных фоновых задач внутри объектов Swing
  - repaint() асинхронный метод, можно вызывать без ограничений
- **ВНИМАНИЕ.** repaint() асинхронный метод, только устанавливает флаг запрос
- **ВНИМАНИЕ.** repaint() при переопределении ОБЯЗАТЕЛЬНО вызывать в первой строке paintComponent()  
`super.paintComponent(g);`
- **ВНИМАНИЕ.** При ДОБАВЛЕНИИ объекта в окно рисования, все надо затирать и рисовать заново
- Способ 1. Создать МАССИВ для хранения объектов, в paintComponent() рисовать из массива при
- Способ 2. Создать IMAGE и дорисовать объект на существующую картинку, Image потом выводить
- Пример. реализации JPanel и getParent().repaint() [lesson\\_ch22/ch/ex34/exercise](#)
- Пример. реализация запуска потоков с визуальными эффектами [lesson\\_ch22/ch/ex34/codea](#)
  - CBox класс JPanel с своей собственной локальной задачей
  - CBoxColors класс JFrame загружает все панели и запускает потоки
- ```
public class CBox extends JPanel implements Runnable { // панель со своим потоком
    private int pause; // локальной фоновой задачи
    private Random rnd = new Random();
    private Color color = new Color(0);
    protected void paintComponent(Graphics g) {
        super.paintComponent(g); // ВНИМАНИЕ. Вызывать ОБЯЗАТЕЛЬНО
        g.setColor(color); // чтобы не было наложений при перерисовке
        Dimension dim = getSize(); // свои размеры
        g.fillRect(0, 0, dim.width, dim.height);
    }
    public CBox(int pause) { // параметр только пауза
        this.pause = pause;
    }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                color = new Color(rnd.nextInt(0xFFFFFF));
                repaint(); // асинхронный только устанавливает флаг запрос на перерисовку
                TimeUnit.MILLISECONDS.sleep(pause);
            }
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
    }
}
public class ColorBoxes extends JFrame {
    private int grid = 12;
    private int pause = 50; //ms
    private static ExecutorService exec = Executors.newCachedThreadPool();
    public ColorBoxes() {
        setLayout(new GridLayout(grid, grid)); // 12x12
        for (int i = 0; i < grid * grid; i++) { // 144 окошка для панелей
            CBox cBox = new CBox(pause);
            add(cBox);
            exec.execute(cBox); // запустили фоновый поток панели
        }
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                exec.shutdownNow(); // закрыть все потоки
                while (!exec.isTerminated()) { // waiting
                }
                super.windowClosing(e);
            });
    }
    public static void check() {
        SwingConsole.run(new ColorBoxes(), 500, 400);
    }
}
```

## JavaBeans

- JavaBeans технология программирования
    - Bean Component просто класс который поддерживает стандартные поля и методы
  - Соглашения JavaBeans о компонентах Bean
    - для поля fieldName задать Getter getFieldName(), для <boolean> вида isFieldName()
    - задать Setter getFieldName()
    - все методы JavaBean public
    - для события EventName определить методы addNameListener(), removeNameListener()
  - Пример. реализация Bean компонента
- [lesson\\_ch22/ch/ex35/codea](#)
- ```
public class Frog {  
    private int jumps;  
    private Color color;  
    private Spot spots;  
    private boolean jumper;  
    public int getJumps() {  
        return jumps;  
    }  
    public void setJumps(int jumps) {  
        this.jumps = jumps;  
    }  
    public Color getColor() {  
        return color;  
    }  
    public void setColor(Color color) {  
        this.color = color;  
    }  
    public Spot getSpots() {  
        return spots;  
    }  
    public void setSpots(Spot spots) {  
        this.spots = spots;  
    }  
    public boolean isJumper() {  
        return jumper;  
    }  
    public void setJumper(boolean jumper) {  
        this.jumper = jumper;  
    }  
    public void addActionListener(ActionListener al) {  
    }  
    public void removeActionListener(ActionListener al) {  
    }  
    public void addKeyListener(ActionListener kl) {  
    }  
    public void removeKeyListener(ActionListener kl) {  
    }  
    // ordinary methods  
    public void croak() {  
        System.out.println("Ribbet!");  
    }  
}
```

## Introspector Получение информации о Bean компоненте

- Introspector класс, который собирает информацию о компоненте Bean
  - работает также как Reflections
  - getBeanInfo() статический метод, который извлекает информацию из компонента Bean
- Пример. реализация Dumper на базе BeanInfo  
[lesson\\_ch22/ch/ex35/codea](#)
- ```
public class BeanDumper extends JFrame {  
    private JTextField query = new JTextField(20);  
    private JTextArea results = new JTextArea();  
    class Dumper implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            String name = query.getText();  
            Class<?> className = null;  
            try {  
                className = Class.forName(name);  
            } catch (Exception ex) {  
                results.setText("Couldn't find " + className);  
                return;  
            }  
            dump(className);  
        }  
    }  
    public BeanDumper() throws HeadlessException {  
        JPanel jPanel = new JPanel();  
        jPanel.setLayout(new FlowLayout());  
        jPanel.add(new JLabel("Qualified bean name:"));  
        jPanel.add(query); // поле запроса  
        add(jPanel, BorderLayout.NORTH);  
        add(new JScrollPane(results)); // TextArea in center BorderLayout in JScrollPane  
        Dumper dumper = new Dumper(); // ActionListener  
        query.addActionListener(dumper); // сработает в TextField по Enter  
        query.setText("ch22.ex35.codea.Frog");  
        dumper.actionPerformed(new ActionEvent(dumper, 0, "")); // обработчик  
    }  
    public void print(String s) {  
        results.append(s + "\n");  
    }  
    public void dump(Class<?> bean) {  
        results.setText(""); // clear TextArea  
        BeanInfo beanInfo = null;  
        String regex = "(\\w+\\.)*";  
        try {  
            beanInfo = Introspector.getBeanInfo(bean, Object.class);  
        } catch (IntrospectionException e) {  
            print("Couldn't introspect " + bean.getName()); // bean Class<?>  
            return;  
        }  
    }  
}
```

- Пример. продолжение <>

lesson\_ch22/ch/ex35/codea

```

•
for (PropertyDescriptor propDescr : beanInfo.getPropertyDescriptors()) { // поля
    Class<?> propClass = propDescr.getPropertyType(); // class поля
    if (propClass == null) {
        continue;
    }
    print("Property type:");
    print(" " + propClass.getName().replaceAll(regex, ""));
    print("Property name: ");
    print(" " + propDescr.getName()); // имя поля
    Method readMethod = propDescr.getReadMethod();
    if (readMethod != null) {
        print("Read method:");
        print(" " + readMethod.getName().replaceAll(regex, "") + "()"); //getter
    }
    Method writeMethod = propDescr.getWriteMethod();
    if (writeMethod != null) {
        print("Write method: ");
        print(" " + writeMethod.getName().replaceAll(regex, "") + "()"); //setter
    }
    print("=====");
}
print("Public methods:");
for (MethodDescriptor methodDescr : beanInfo.getMethodDescriptors()) {
    print(methodDescr.getMethod().toString().replaceAll(regex, ""));
}
print("=====");
print("Event support:");
for (EventSetDescriptor eventDescr : beanInfo.getEventSetDescriptors()) {
    print("Listener type:");
    print(" " + eventDescr.getListenerType().getName().replaceAll(regex, ""));
    for (Method method : eventDescr.getListenerMethods()) {
        print("Listener method:");
        print(" " + method.getName().replaceAll(regex, "") + "()");
    }
    for (MethodDescriptor methodDescr :
            eventDescr.getListenerMethodDescriptors()) {
        print("Method descriptor:");
        print(" " +
            methodDescr.getMethod().toString().replaceAll(regex, "") + "()");
    }
    Method addListener = eventDescr.getAddListenerMethod();
    print("Add Listener Method:");
    print(" " + addListener.toString().replaceAll(regex, "") + "()");
    Method removeListener = eventDescr.getRemoveListenerMethod();
    print("Remove Listener Method:");
    print(" " + removeListener.toString().replaceAll(regex, "") + "()");
    print("=====");
}
public static void check() {
    SwingConsole.run(new BeanDumper(), 600, 500);
}
}

```

## Bean Сложные компоненты

- Bean Сложные компоненты
  - используются данные Bean компонента для динамической обработки
- Пример. реализация сложного компонента Bean
  - поля            все поля имеют Getter и Setter
  - Listeners      есть методы addListeners и removeListeners
  - если Listener добавлен, то он будет обработан по нажатию кнопки мыши
  - в данном классе можно подключить только один Listener
  - Serializable    среда Swing может сохранить свойства объекта
  -

[lesson\\_ch22/ch/ex35/codeb](#)

```
public class BangBean extends JPanel implements Serializable {  
    private int xm;  
    private int ym;  
    private int cSize = 20;  
    private String text = "Bang!";  
    private int fontSize = 48;  
    private Color tColor = Color.RED;  
    private ActionListener actionListener;  
    class ML extends MouseAdapter {  
        @Override  
        public void mousePressed(MouseEvent e) {  
            Graphics g = getGraphics();  
            g.setColor(tColor);  
            g.setFont(new Font("TimeRoman", Font.BOLD, fontSize)); // задать новый фонт  
            int width = g.getFontMetrics().stringWidth(text); // ширина шрифта  
            g.drawString(text, (getSize().width - width) / 2, getSize().height / 2);  
            g.dispose(); //в центре кадра  
            if (actionListener != null) { // если существует то вызвать авто сработку  
                actionListener.actionPerformed(new ActionEvent(  
                    BangBean.this, ActionEvent.ACTION_PERFORMED, null)  
            );  
        }  
    }  
    class MML extends MouseMotionAdapter {  
        @Override  
        public void mouseMoved(MouseEvent e) { // обновляются координаты  
            xm = e.getX();  
            ym = e.getY();  
            repaint(); //не вызывается головной объект т.к. есть super.paintComponent  
        }  
    }  
    public BangBean() {  
        addMouseListener(new ML());  
        addMouseMotionListener(new MML());  
    }  
    @Override  
    public Dimension getPreferredSize() {  
        return new Dimension(200, 200);  
    }  
    @Override  
    protected void paintComponent(Graphics g) { // перерисовка окружности на новом месте  
        super.paintComponent(g); // чтобы использовать локальный repaint()  
        g.setColor(Color.BLACK);  
        g.drawOval(xm-cSize/2,ym-cSize/2,cSize,cSize); // окружность вокруг центра  
    }  
}
```

- Пример. продолжение <>

[lesson\\_ch22/ch/ex35/codeb](#)

```
//JavaBeans support
public int getSize() {
    return cSize;
}
public void setSize(int cSize) {
    this.cSize = cSize;
}
public String getText() {
    return text;
}
public void setText(String text) {
    this.text = text;
}
public int getFontSize() {
    return fontSize;
}
public void setFontSize(int fontSize) {
    this.fontSize = fontSize;
}
public Color getColor() {
    return tColor;
}
public void setColor(Color tColor) {
    this.tColor = tColor;
}
// JavaBean Listeners
public void addActionListener(ActionListener al) throws TooManyListenersException{
    if (actionListener != null) {
        throw new TooManyListenersException(); // обработчик может быть только один
    }
    actionListener = al; // получить свой обработчик
}
public void removeActionListener(ActionListener al){
    actionListener = null; // отключить обработчик
}
}
```

- Пример. продолжение реализация BangBeanTest

[lesson\\_ch22/ch/ex35/codeb](#)

```
public class BangBeanTest extends JFrame {
    private JTextField jTextField = new JTextField(20);
    class BBL implements ActionListener {
        private int count = 0;
        @Override
        public void actionPerformed(ActionEvent e) {
            jTextField.setText("BangBean action " + count++);
        }
    }
    public BangBeanTest() throws HeadlessException {
        BangBean bangBean = new BangBean();
        try {
            bangBean.addActionListener(new BBL());
        } catch (TooManyListenersException e) {
            jTextField.setText("Too many Listeners");
        }
        add(bangBean); // добавить в центр BorderLayout
        add(jTextField, BorderLayout.SOUTH); // текстовое поле внизу окна
    }
    public static void check() {
        SwingConsole.run(new BangBeanTest(), 400, 500);
    }
}
```

-

## Построение GUI на базе объектов JavaBean

- Построение GUI на базе объектов JavaBean
    - создается форма в IntelliJ IDEA, задается имя главной панели
    - добавляется Non-Palette объект, также задается его имя, выбирается класс BangBean
    - в классе GUI добавляется расширение от JFrame и добавляется главная панель
    - класс GUI запускается стандартными средствами
  - ВНИМАНИЕ. Размеры компонентов задаются в форме PreferredSize либо в методе getPreferredSize()
  - Пример. реализация GUI с BangBean
- [lesson\\_ch22/ch/ex35/exercise](#)
- ```
public class JavaBeanGUI extends JFrame {  
    private JTextField textField1; // компоненты формы в GUI IntelliJ IDEA  
    private JCheckBox checkBox1;  
    private JCheckBox checkBox2;  
    private JCheckBox checkBox3;  
    private JPanel rootPanel;  
    private BangBean bangBean; // custom class для GUI IntelliJ IDEA  
    public JavaBeanGUI() {  
        checkBox2.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                System.out.println("JavaBeanGUI.actionPerformed");  
            }  
        });  
  
        add(rootPanel);  
    }  
    public static void check() {  
        SwingConsole.run(new JavaBeanGUI(), 500, 500);  
    }  
}
```

## JavaBean Компоненты и Синхронизация

- JavaBean Компоненты в Многозадачном режиме
- Правила работы JavaBean в многозадачном режиме
  - все public методы должны быть synchronized
  - при запуске многоадресного Listener учесть изменение числа объектов которые слушают событие
- Работа с мноадресным Listener
  - метод            synchronized void addActionListener()
  - метод            synchronized void removeActionListener()
  - метод            notificationListeners()
  - synchronized{} блок клонирования списка Listeners
  - рассылка ActionEvent только из клона Listeners
- Работа с paintComponent()
  - метод            paintComponent() также должен быть синхронизирован
- Критерии оценки методов для работы с synchronized
- Доступ к критическим переменным
  - определить изменяет ли метод состояние критических переменных объекта
  - если да, то метод НАДО синхронизировать
  - **ВНИМАНИЕ.** критические переменные это те которые читаются и пишутся из разных потоков
  - (paintComponent() не использует критические переменные)
- Зависимость от критических переменных
  - определить зависит ли работа метода от критических переменных
  - если другой синхронный метод меняет критическую переменную от которой зависит данный
  - если да то метод НАДО синхронизировать
  - (paintComponent() будет зависеть от cSize если ее будет менять метод из другого потока)
- Синхронность методов базового класса
  - определить синхронизирован ли такой же метод у базового класса
  - если да, то да НАДО синхронизировать
  - (paintComponent() у базового класса не синхронизирован)
  - **ВНИМАНИЕ.** синхронизация НЕ передается по наследству, поэтому это может быть проблемой
- Время выполнения
  - определить насколько критична скорость выполнения методов
  - если скорость важна, то синхронизация может стать серьезным препятствием
  - если да, то ЛУЧШЕ пересмотреть структуру программы и использовать несинхронные методы
  - (paintComponent() несинхронный как раз потому что требуется высокая скорость)

- Пример. реализация многоадресного Listener и synchronized

[lesson\\_ch22/ch/ex36/codea](#)

- multiaddress List<ActionListener>
  - synchronized addActionListener()
  - synchronized removeActionistener()
  - synchronized paintComponent()

```

• public class BangBean2 extends JPanel implements Serializable {
    private int xm;
    private int ym;
    private int cSize = 20;
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;           // МНОГОАДРЕСНЫЙ слушатель
    private ArrayList<ActionListener> actionListener = new ArrayList<>();

    class ML extends MouseAdapter {
        @Override
        public void mousePressed(MouseEvent e) {
            Graphics g = getGraphics();
            g.setColor(tColor);
            g.setFont(new Font("TimeRoman", Font.BOLD, fontSize)); // задать новый ФОНТ
            int width = g.getFontMetrics().stringWidth(text); // ширина шрифта
            g.drawString(text, (getSize().width - width) / 2, getSize().height / 2);
            g.dispose();
        }
        // actionListener.actionPerformed(new ActionEvent(
        //     BangBean2.this, ActionEvent.ACTION_PERFORMED, null));
        notifyListeners(); // вместо одного рассыпаем событие всем
    }

    class MML extends MouseMotionAdapter {
        @Override
        public void mouseMoved(MouseEvent e) { // обновляются координаты
            xm = e.getX();
            ym = e.getY();
            repaint(); //не вызывается головной объект т.к. есть super.paintComponent
        }
    }

    public BangBean2() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(200, 200);
    }

    @Override
    protected synchronized void paintComponent(Graphics g) { // перерисовка окружности
        super.paintComponent(g); // чтобы использовать локальный repaint()
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize / 2, ym - cSize / 2, cSize, cSize); // окружность в центре
    }
}

```

- Пример. продолжение<>

[lesson\\_ch22/ch/ex36/codea](#)

```
//JavaBeans support
public int getSize() {
    return cSize;
}
public void setSize(int cSize) {
    this.cSize = cSize;
}
public String getText() {
    return text;
}
public void setText(String text) {
    this.text = text;
}
public int getFontSize() {
    return fontSize;
}
public void setFontSize(int fontSize) {
    this.fontSize = fontSize;
}
public Color getColor() {
    return tColor;
}
public void setColor(Color tColor) {
    this.tColor = tColor;
}
// JavaBean Listeners ВНИМАНИЕ многоадресный слушатель
public synchronized void addActionListener(ActionListener al) {
    actionListener.add(al); // получить свой обработчик
}
public synchronized void removeActionListener(ActionListener al) {
    actionListener.remove(al); // отключить обработчик
}
public void notifyListeners() {
    ActionEvent action = new ActionEvent(
        BangBean2.this, ActionEvent.ACTION_PERFORMED, null);
    ArrayList<ActionListener> clone = null;
    synchronized (this) { // блок синхронизации
        clone = actionListener; // для работы с клоном но он копия только по ссылке
    }
    for (ActionListener listener : clone) { // по сути рассылка события
        listener.actionPerformed(action); // перебрать Listener и все оповестить
    }
}
public static void check() {
    BangBean2 bangBean2 = new BangBean2();
    bangBean2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Action Event " + e);
        }
    });
    bangBean2.addActionListener(new ActionListener() { // второй Listener
        public void actionPerformed(ActionEvent e) {
            System.out.println("BangBean2 action ");
        }
    });
    bangBean2.addActionListener(new ActionListener() { // третий Listener
        public void actionPerformed(ActionEvent e) {
            System.out.println("More action ");
        }
    });
    JFrame jFrame = new JFrame(); // запуск JFrame снаружи
    jFrame.add(bangBean2);
    SwingConsole.run(jFrame, 300, 300);
}
```

## Упаковка Bean в JAR

- Упаковка Bean в JAR текущее состояние
  - сейчас уже нет особой разницы, во всех IDE идет просто работа с классами
  - NetBeans называет классы Bean
  - IntelliJ IDEA называет классы Non-Palette
  -
- Упаковка Bean для работы с GUI это просто упаковка класса в JAR архив
  - Bean упаковывается в JAR архив с файлом MANIFEST, где указано что это объект Bean
  - MANIFEST при упаковке должен быть на уровне src от которого отрабатывает package файлов class
  -
- Пример. реализация JAR упаковки с файлом MANIFEST для объекта bangbean.BangBean  
jar cfvm BangBean.jar BangBean.mf bangbean
- 
- **ВНИМАНИЕ.** расширение mf для манифеста ОБЯЗАТЕЛЬНО
- Пример. реализация файла MANIFEST для объекта bangbean.BangBean где package bangbean  
Manifest-Version: 1.0  
Name: bangbean/BangBean.class  
Java-Bean: True  
<еще одна пустая строка обязательно>
- **ВНИМАНИЕ.** В общем это только для NetBeans актуально, в IntelliJ IDEA class добавляется как Non-Palette

## Создание JAR из класса в IDEA

- Пример. реализация создания JAR архива прямо из среды IntelliJIDEA [lesson\\_ch22/ch/ex36/codeb](#)
  - файл компилируется как обычно
  - затем запускается Exercise и командный файл прямо из среды создает JAR файл
- Пример. реализация файла BangBean3 [lesson\\_ch22/ch/ex36/codeb](#)
- Пример. реализация файла jarc.cmd [lesson\\_ch22/ch/ex36/codeb](#)  
rem .\src\ch22\ex36\exercise\jarc.cmd  
cd .\src\ch22\ex36\exercise\  
rmdir /q /s ch22\ex36\exercise  
del \*.jar  
mkdir ch22\ex36\exercise  
copy ..\..\..\out\production\jb01\ch22\ex36\exercise\b\*3\*.class  
.\\ch22\ex36\exercise\  
jar cvfm BangBean3.jar BangBean3.mf ch22/ex36/exercise  
cd ..\..\..\..
- 
- Пример. реализация файла BangBean3.mf [lesson\\_ch22/ch/ex36/codeb](#)
- **Manifest-Version:** 1.0  
**Name:** ch22/ex36/exercise/BangBean3.class  
**Java-Bean:** True
- Применение. [lesson\\_ch22/ch/ex36/codeb](#)
- ```
public static void app() {  
    System.out.println("\n=====EXERCISE====");  
    System.out.println("\nExercise Check\n");  
    //  
    CmdExec.run(". \\src\\ch22\\ex36\\exercise\\jarc2.cmd");  
    try {  
        Process p = new ProcessBuilder("cmd /c .\\src\\ch22\\ex36\\exercise\\\"+  
   "jarc.cmd".split(" ")).inheritIO().start(); // создать новый процесс  
        p.waitFor();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    JFrameGUI.check();  
}
```

## Работа с формами и JAR класса Bean в IDEA

- Работа с формами в IDEA
  - создается форма, размещаются компоненты Swing
  - добавляется архив JAR в библиотеку проекта, чтобы был доступен класс
  - **ВНИМАНИЕ.** Здесь работа будет идти с локальным классом, с классом из JAR все тоже самое
  - подключается класс и размещается на форме
  - добавляются изменения в исходник формы до класса JFrame
  - форма готова к работе
- Создание формы IDEA
  - File >> New >> GUI Form >> ввести имя класса >> «JFrameGUI»
  - Select JPanel >> field name >> ввести имя главной панели >> «jPanel»
  - добавить любые компоненты на форму внутрь панели, разместить по желанию
  - задать имя главной панели вручную или провести binding автоматически
- **ВНИМАНИЕ.** Чтобы вручную задать имя любого объекта формы надо внести поле с именем JFrameGUI
- выбрать компонент и ввести в поле fieldName то же имя поля что в исходнике JFrameGUI
- Подключение класса Bean
  - Select Non-Pallette >> Click Form в пустом месте >> ввести имя >> BangBean3 >> Check Create binding
  - Select BangBean3 >> Properties >> Preferred Size >> 500,400
- Изменения в исходнике формы
  - добавить расширение JFrame >> « class JFrameGUI extends JFrame»
  - добавить конструктор
  - добавить вызов add(jPanel)
- Все, форма загружается как обычно командой SwingConsole.run(new Jframe, width, size)
- Пример. реализация формы
- ```
public class JFrameGUI extends JFrame{ // добавлено вручную extends JFrame
    private JPanel jPanel;
    private JButton button1;
    private BangBean3 bangBean31;

    public JFrameGUI() throws HeadlessException { // добавлено вручную конструктор
        add(jPanel);
    }
    public static void check() { // добавлено вручную вызов JFrame
        SwingConsole.run(new JFrameGUI(), 500, 400);
    }
}
```

[lesson\\_ch22/ch/ex36/codeb](#)

## Работа с формами и JAR класса Bean в NetBeans

- Работа с формами в NetBeans
  - создается форма JFrame , размещаются компоненты Swing
  - ВНИМАНИЕ. ОБЯЗАТЕЛЬНО выбрать форму JFrame тогда создается сразу main() внутри формы
  - импортируется я архив JAR в библиотеку компонентов Bean, чтобы был доступен класс
  - компонент ласс и размещается на форме
  - добавляются изменения в исходник формы до класса JFrame
  - форма готова к работе, вызвать main()
- Создание формы
  - File >> New Project >> JavaGUI,
  - File >> new File >> Swing GUI Forms >> JFrame Form>>Class name >> «JFrameGUI»
- Импорт компонента
  - Tools >> Pallette >> Swing/AWT Components >> From JAR >> Select JAR >> Select BeanBang3
  - >> Select «Swing Containers» >> Add BangBean3 to Swing Containers
  - Компонент BangBean3 теперь появится в панели инструментов Swing Containers
  - Select BangBean3 >> Place on Form
  - Select JFrame >> RMB >> Preview Design >> Metal проверить что форма работает с BangBean3
- Подключение формы к проекту
  - JFrame уже имеет встроенный main() и нормально запускается
  - Добавить вызов JFrameGUI.main() в главный класс проекта
- Пример. реализация Bean в NetBeans
- ```
class JavaGUI {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        JFrameGUI.main(args);  
    }  
}
```
- 

## Работа с исходниками JFrame GUI

- Работа с исходниками JFrame GUI lesson\_ch22/ch/ex36/exercise
  - Bean класс должен наследовать любой компонент Swing
  - Frog не наследует JComponent, NetBeans его распознал на форме не разместил
  - BangBean4 наследует JFrame и все нормально разместилось
  - Манифест может содержать описание нескольких классов  
**Manifest-Version:** 1.0  
**Name:** ch22/ex36/exercise/jbean/BangBean4.class  
**Java-Bean:** True  
**Name:** ch22/ex36/exercise/jbean/Frog.class  
**Java-Bean:** True
  -
- ВНИМАНИЕ. Class Bean ОБЯЗАТЕЛЬНО должен наследовать JComponent то есть любой Swing

## Добавление Bean компонентов на форму

- Добавление Bean компонентов на форму
  - создается обычный класс
  - на все поля добавляются Getter и Setter
  - класс наследуются от любого компонента Swing, то есть потомка JComponent
  - в классе переопределяется метод getPreferredSize() и задается размер, в форме будет этот размер
  - класс компилируется, для чего надо добавить объект в main() и запустить
  - класс добавляется на GUI форму, ему назначается имя вручную или <binding auto>
  - классу можно задать цвет поля прямо в форме и посмотреть это поле при вызове формы

- Пример. реализация добавления компонента Valve на форму

[lesson\\_ch22/ch/ex37/exercise](#)

```
public class Valve extends JPanel {  
    boolean state = false;  
    int level = 0;  
    public Valve() {  
        state = true;  
        level = 50;  
    }  
    public boolean isState() {  
        return state;  
    }  
    public void setState(boolean state) {  
        this.state = state;  
    }  
    public int getLevel() {  
        return level;  
    }  
    public void setLevel(int level) {  
        this.level = level;  
    }  
    @Override  
    public Dimension getPreferredSize() {  
        return new Dimension(250, 100);  
    }  
}
```

- Пример. реализация добавления компонента Valve на форму

[lesson\\_ch22/ch/ex37/exercise](#)

```
public class JFrameVGUI extends JFrame {  
    private JButton button1;  
    private JButton button2;  
    private JCheckBox checkBox1;  
    private JCheckBox checkBox2;  
    private Valve valve;  
    private JPanel jPanel;  
  
    public JFrameVGUI() throws HeadlessException {  
        add(jPanel);  
    }  
  
    public static void check() {  
        SwingConsole.run(new JFrameVGUI(), 600, 500);  
    }  
}
```

## Пропущенные разделы

- Пропущенные разделы
  - Flash              технология отменена в марте 2017 года
  - SWT                технология предшественний Swing устарела

## Заключение

- Это заключительная страница основного текста

## Список приложений

- Далее идут приложения
- Приложение A              Работа в IDEA              описание рутинных процедур
- Приложение B              Hints                      описание приемов
- Приложение C              Нерешенные вопросы      то что так и осталось неясным
- Приложение D              Примеры реализации    большие исходники, чтобы не загромождать текст
- Приложение E              Drafts                      временные тексты исходников

## Работа в IDEA

- Подключение модуля
  - распаковать исходники в []sandbox/code
  - подключить модуль к проекту File > New > New Module from Existing Sources > []sandbox/code
  - проверить что исходники нормально работают создать проект, применить класс из /code

## Работа с Java

- Компиляция
  - допустим файл находится в папке src/ch/ex21/local/TestBall.java
  - компилировать из папки ./src > javac ch/ex21/local/TestBall.java
  - будет создан файл TestBall\$Main.class и другие файлы классов
- Запуск
  - запускать из папки ./src > java ch/ex21/local/TestBall\$Main        все должно заработать
- Запуск файлов созданных IDEA
  - запускать из папки ./out/production/lesson\_ch10/ > java ch/ex21/local/TestBall\$Main
  - также все должно сработать
- **ВНИМАНИЕ.** –sourcepath src это указание для package, головной файл задавать с ПОЛНЫМ путем
- javac –sourcepath src src/ex02/Ex02.java
- Компиляция
  - –d src/bin        путь куда запишутся все файлы со всей иерархией каталогов

## CLASSPATH Java

- Компиляция
  - при компиляции CLASSPATH используется для поиска исходников ДОПОЛНИТЕЛЬНО к главному
  - если указан –sourcepath то используется он,
  - если не указан –sourcepath то используется CLASSPATH
- **ВНИМАНИЕ.** Путь к исходникам ДОПОЛНИТЕЛЬНЫМ к главному, путь к главному надо указать явно
- Выполнение
  - CLASSPATH задает путь к файлам \*.class, можно указать путь КО ВСЕМ файлам в разных папках
- **ВНИМАНИЕ.** Путь к классам работает четко, и для всех, и основного и дополнительных
- Пример. реализация теста на CLASSPATH
  - // при компиляции используется –sourcepath
  - set classpath=D:\temp2\cp\out                            // задать некий CLASSPATH
  - javac –sourcepath src in\Main.java                    // при компиляции используется –sourcepath
  - // при компиляции используется CLASSPATH
  - set classpath=D:\temp2\cp\src
  - javac in\Main.java -d D:\temp2\cp\out
  - // при выполнении CLASSPATH указаны не все пути (нет ./src)
  - java Main
  - // при выполнении CLASSPATH указаны все пути
  - set classpath=D:\temp2\cp\out;./src
  - java Main
- Архив, распаковать, читать readme.txt



- cp.zip

## Работа с Annotation

- Работа с Annotation apt
  - путь до корневого каталога class path\_out out/production/jb01tmp
  - путь до корневого каталога java src src
  - команда для
    - процессора out/production/jb01tmp/ex02/codeb/ IFEPPFact.class
    - исходного файла с аннотацией src/ex02/codeb/Mult.java
    - выходной папки src/ex02/codeb
- **ВНИМАНИЕ.** путь до класса процессора задается как сумма –cp path и package\_path самого процессора
  - запуск из каталога []\_sandbox результат сохраняется в jb01tmp/src
  - D:\\_lessons\java\[]\_sandbox>
  - apt -factory ex02.codeb.IFEPPFact ./jb01tmp/src/ex02/codeb/Mult.java -s ./jb01tmp/src -cp jb01tmp/out/production/jb01tmp
  - запуск из каталога jb01tmp результат сохраняется в jb01tmp/src
  - D:\\_lessons\java\[]\_sandbox\jb01tmp>
  - apt -factory ex02.codeb.IFEPPFact ./src/ex02/codeb/Mult.java -s ./src -cp out/production/jb01tmp

## Работа с Annotation javac

- Вызов javac <source> -processor <proc> -d <pathOutClass> -s <pathOutJava> -cp <pathClasses>
  - <source> файл \*.java который будет скомпилирован и обработан процессором
  - <proc> файл \*.class annotation processor который будет обрабатывать исходники
  - <pathOutClass> путь куда запишутся файлы компиляции class
  - <pathOutJava> путь куда запишутся сгенерированные процессором файлы java
  - <pathClasses> путь до корневой папки package файлов классов
- Пример.
  - путь до корневого каталога class out/production/jb01tmp
  - путь до корневого каталога java src
  - команда для
    - процессора out/production/jb01tmp/ex02/codeb/ IFEProcC.class
    - исходного файла с аннотацией src/ex02/codeb/Mult.java
    - выходной папки src/
  - запуск из каталога jb01tmp результат сохраняется в jb01tmp/src
  - D:\\_lessons\java\[]\_sandbox\jb01tmp>javac ./src/ex02/codec/Mult.java -d src -cp out/production/jb01tmp -processor ex02.codec.IFEProcC
- Код файла процессора IFEProcC.java
  - обязательно должен содержать тип аннотаций и версию компиляции
- ```
@SupportedAnnotationTypes({ "ex02.codecEIFace74", "ex02.codecEIFace75" })
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class IFEProcC extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> an, RoundEnvironment env) {
        System.out.println("javac Annotation Processor run");
        return true;
    }
}
```
- Код анализируемого файла Mult.java
  - `@EIFace75("IMultiplier75") // назначение вытащить открытый интерфейс класса`
  - `public class Mult { }`

## Работа с Ant



# JavaDoc

- JavaDoc            компиляция документации
  - 
  - **ВНИМАНИЕ.** Для компиляции русских символов ввести в окне Other arguments команду «-encoding utf8»

## JAR создание и подключение в IDEA

- Создание JAR смотреть в главе ch22ex36/codeb, ch22/ex36/exercise
    - полное описание и примеры для создания простого JAR
    - полное описание и примеры для создания JavaBeans JAR
    -

## Сравнение примитивов и объектов

- **ВНИМАНИЕ.** При переопределении equals обязательно запустить механизм override

## Логические операции

- **ВНИМАНИЕ.** ~ инверсия работает с целыми числами
  - по принципу дополнения инвертировать знак и вычесть 1
  - $\sim -2 \Rightarrow 2 - 1 \Rightarrow 1$
  - $\sim 0 \Rightarrow 0 - 1 \Rightarrow -1$  неочевидно, но да, инверсия от 0 это  $-1$
  - $\sim -1 \Rightarrow 1 - 1 \Rightarrow 0$  неочевидно, но да, инверсия от  $-1$  это 0

## Что надо помнить при инициализации

- Нельзя создавать static объекты в конструкторе класса
  - **ВНИМАНИЕ.** static НЕЛЬЗЯ инициализировать КОНСТРУКТОРЕ создает МУСОР и ПОТЕРЮ ДАННЫХ
  - создание мусора новый экземпляр пересоздает static объект, выбрасывает предыдущий
  - потеря данных новый экземпляр пересоздает static объект, данные предыдущего теряются
- 

## Работа с массивами Object

- Автоматическое определение типа Object[]
  - <Object ...> позволяет принять в метод массив Object[] и автоматически распознать тип

## Override toString

[lesson\\_ch7/ch/ex1/Plate](#)

- классная функция, позволяет просто бросать объект в строку и вывод будет какой задан в этом методе

## Завершение программы

- **ВНИМАНИЕ.** Использовать секцию finally, которая выполняется ВСЕГДА, до закрытия программы
- Приоритет перед finally есть только у команды System.exit(0)

## Полиморфные методы при вызове из конструкторов

- Вызов переопределенного метода в конструкторе предка ведет к ошибке [lesson\\_ch08/ch/ex16](#)
  - потому что вызывается метод потомка, и данные потомка, которые еще не инициализированы
  - **ВНИМАНИЕ.** переопределенные методы в конструкторе предка работают с не готовыми данными

## Локальные внутренние классы

- локальные внутренние классы могут создаваться в блоках кода или метода
- локальный класс имеет доступ к final текущего блока, метода, а также ко всем полям внешнего класса
- **ВНИМАНИЕ.** Локальные классы применяют вместо анонимных, когда нужно несколько экземпляров

## Многомерные массивы

- Многомерные массивы
  - все измерения могут быть разной длины
  - **ВНИМАНИЕ.** печать многомерным массивов Arrays.deepToString()

## Hints

### Внутренние классы размещение в разных областях методах, классах и прочее

- Области где можно создать и использовать внутренний класс
  - внутри метода
  - в области действия {} внутри метода
  - анонимный класс, реализованный в интерфейсе ???
  - анонимный класс, внутри пользовательского конструктора класса ???
  - анонимный класс выполняющий инициализацию поля ???
  - анонимный класс выполняющий конструирование и инициализацию экземпляра

### Внутри метода

- Внутренний класс внутри метода
  - может быть определен внутри всего метода
  - может быть определен внутри части области видимости внутри метода например if{} else{}

[lesson\\_ch10/ch/ex9 /ex10](#)

[lib/\\_inner\\_classes/ex1](#)

- Пример. Весь метод

```
public class Recorder {  
    public ITape getTape() { // method create object inside it  
        class Tape implements ITape {  
            @Override  
            public void showInfo() {  
                System.out.println("Tape.showInfo");  
            }  
        }  
        return new Tape();  
    }  
}
```

- Пример. Внутри ограниченной области видимости внутри метода, например в области условия if()

```
public class Tape {  
    public IPlay getTape(boolean flag) { // method create object inside it  
        if (flag) {  
            class Record implements IPlay { // область видимости scope {flag = true}  
                @Override  
                public void play() {  
                    System.out.println("Record.play");  
                }  
            } // eof class  
            return new Record();  
        } else {  
            class MidiBoard implements IPlay { // область видимости scope {flag = false}  
                @Override  
                public void play() {  
                    System.out.println("MidiBoard.play");  
                }  
            } // eof class  
            return new MidiBoard();  
        }  
    }  
}
```

### Автоидентификация объектов класса

- Каждый объект класс содержит уникальный идентификатор создаваемый автоматом
  - counter создается один раз при объявлении класса
  - далее наращивается с каждым объектом и идентификатор фиксированный

```
public class Apple { // объект хранения  
    private static long counter;  
    private final long id = counter++; // автоинидентификация при создании объектов  
}
```

## Hints

### Добавление групп элементов

- Collection и Arrays есть методы добавления группы элементов [lesson\\_ch11/ch/ex4/local/include](#)
- Arrays.asList() принимает массив или список элементов и создает Collection или List объект
- ВНИМАНИЕ.** Методы добавления группы создания списка через Collections или через кастинг в asList()

```
List<Snow> snow3 = new ArrayList<>(); // создаем объект List все объекты extends Snow
Collections.addAll(snow3,new Light(), new Heavy(), new Crusty()); // через collection
```

```
List<Snow> snow4 = Arrays.<Snow>asList( // создаем объект List все объекты extends Snow
    new Light(), new Heavy(), new Powder(), new Crusty(), new Slush()
); // через явное указание типа
```

### ListIterator

- Работает только с List
- ВНИМАНИЕ.** для установки итератора в конец списка надо СОЗДАТЬ его с параметром list.size()

### Format String in Columns

- Печать слов в несколько колонок с заданным отступом
- %12s, %12d ширина 12 символов, выравнивание по правому краю, работает для String, Integer
- %-12s, %-12d тоже самое выравнивание по левому краю, , работает для String, Integer
- System.out.printf(index++ + ": %-10s %2d\n", m.group(), m.start());
- System.out.printf(index++ + ": %-10s %-2d\n", m.group(), m.start());
- 
- Пример

```
System.out.println("Print using Format %%12s to Split on Columns\n");
Iterator<String> it = tset.iterator();
for (int i = 0; i < tset.size(); i++) { // крутим пока есть индексы
    System.out.printf("%-12s",it.next());
    if ((i > 0) && ((i % 8) == 0)) {
        System.out.println(); // каждое 25 слово переводим строку
    }
}
```

### Set вывод на печать класс хранения пользователя

- Вывод на печать класс хранения пользователя в три этапа [lesson\\_ch11/ch/ex16/access](#)
  - добавить свою функцию toString в класс пользователя с форматом вывода
  - использовать свою функцию при выводе экземпляра класса через итератор

```
public class Word {
    private String word;
    private int count;

    public String toString(String format) {
        return String.format(format,word,count);
    }
    @Override
    public String toString() {
        return word + "{" + count + '}';
    }
}

public static void showSet(Set<Word> set) {
    Iterator<Word> it = set.iterator();
    int k = 1;
    while (it.hasNext()) {
        System.out.printf(it.next().toString("%-14s:%2d    "));
        if (((k++ % 8) == 0)) {
            System.out.println(); // каждое 25 слово переводим строку
        }
    }
}
```

### Hints

- **ВНИМАНИЕ.** Если не проверять Comparator, то рассматривает как разные объекты по ссылкам даже если значения внутри класса одинаковые

## Разница между printStackTrace и fillInStackTrace

- printStackTrace() распечатывает весь массив кадров стека
- **ВНИМАНИЕ.** чтобы не мешался вывод на печать можно подменить вывод printStackTrace с err на out
- Пример

```
try {
    f();
} catch (Exception e) {
    e.printStackTrace(System.out);           //распечатка стека Exception в System.out
}
```
- fillInStackTrace() вытаскивает массив кадров стека для текущей точки перехвата Exception
- **ВНИМАНИЕ.** fillInStackTrace() это ОБРАЗ СТЕКА В ТЕКУЩЕЙ ТОЧКЕ можно использовать как новый адрес для генерации exception
- Пример.

```
try {
    f();
} catch (Exception e) {
    e.printStackTrace(System.out);           //распечатка стека Exception в System.out
    throw (Exception)e.fillInStackTrace();  //создание Exception с новым контекстом
}
```

[lesson\\_ch12/ch/ex10/include](#)

## StringBuilder

- StringBuilder используют в реализации метода toString() для сложных объектов

- Пример.

```
@Override
public String toString() {
    StringBuilder stringB = new StringBuilder();
    for (Object[] field : fields) {           // прогнать все элементы
        stringB.append(field[0].toString());     // каждый элемент распаковать в
    }
    stringB.append(" : ");
    stringB.append(field[1].toString());
    stringB.append("\n");
}
return stringB.toString();                   // выдать stringB в строку
}
```

## Matcher Groups

- Matcher методы работы с группами так
- **ВНИМАНИЕ.** при работе с groupCount() использовать МЕНЬШЕ или РАВНО в цикле
- **ВНИМАНИЕ.** Matcher.find(0) включает все. Matcher.find(1) включает только [lesson\\_ch12a/ch/ex15/local](#) группу 1 можно использовать для отсечения маркеров, если вынести их за пределы группы

[lesson\\_ch12a/ch/ex12/include](#)

## Расширение типа Интерфейса

- Стандартная реализация

```
private static List<IFactory> pList = new ArrayList<>(); // IFactory это интерфейс
```
- Расширенная реализация

```
private static List<IFactory<?extends Part>> pList = new ArrayList<>(); // IFactory<T>
```
- **ВНИМАНИЕ.** List<IFactory<?extends Part>> pList означает Class или Interface с Generic <T>
  - IFactory<T> содержит Generic <T> <T> == ?<extends> Part>
  - поэтому IFactory<T> == ?<extends> Part>>

## Hints

### Рекурсивное вытаскивание информации о класса

- Пример. мощный пример рекурсивной обработки данных класса

[lesson\\_ch14/ch/ex20/access](#)

### toString() параметры объекта

- toString() параметры объекта выводится имя объекта @hashCode().HEX  
`return this.getClass().getName() + "@" + Integer.toHexString(this.hashCode());`

### Доступ к закрытым членам класса

- Доступ к закрытым членам класса
  - зная имя метода и используя метод setAccessible() можно запустить даже private методы
  - узнать имена методов класса можно при помощи декомпилиатора javap -private
- ВНИМАНИЕ.** javap -private и метод Method.setAccessible() дают доступ к закрытым методам класса
- Пример.

### SortedSet Интерфейс реализация

- ВНИМАНИЕ.** классный пример
- Пример. SortedList реализация Sorted дополнения LinkedList CheckSorted.app()
- Пример. SortedLinked реализация SortedSet интерфейса с Comparator CheckSorted.app2()

### Hash for Speed

- ВНИМАНИЕ.** мощные примеры
- Примерный алгоритм поиска по hashCode()
  - весь массив значений разбивается на подмассивы меньшего размера
  - по ключу строится hashCode который указывает адрес такого подмассива
  - далее поиск в подмассиве идет «медленным способом» методом equals()
  - скорость достигается за счет быстрого получения hash code и размера подмассива 2..3 элемента
- Пример. простая реализация SimpleHashMap с полным набором команд [lesson\\_ch17/ch/ex23/access](#)
  - особенно интересно индексирование по размеру таблицы (hashCode()%SIZE)
- Пример. простая реализация SimpleHashSet с полным набором команд [lesson\\_ch17/ch/ex24/access](#)
- Пример. Полная реализация SimpleHashMapL на своих EntrySet и LinkEntry [lesson\\_ch17/ch/ex25/access](#)

### Выбор и тестирование с List

- Пример. подключение методов ArrayList и LinkedList одновременно
- ВНИМАНИЕ.** Общий смысл вставить семафор и пересоздавать ArrayList | LinkedList при переключении

### Выбор и тестирование с Map

- Работа и тестирование с Map
- Пример. реализация SlowMapEntry на List<Entry<K,V>> с компаратором [lesson\\_ch17/ch/ex35/codea](#)
- Пример. реализация SlowMapEntrySort на базе List<Entry<K,V>> с сортировкой, резко ускорен метод get()
- ВНИМАНИЕ.** Мощные примеры используют неполное сравнение и компаратор в объекте хранения
  - чтобы реализовать Map на базе ArrayList<Entry<K,V>>

### Hints

## Факторы влияющие на производительность HashMap

- Пример. Получение Load Factor и тестирование HashMap с разной загрузкой [lesson\\_ch17/ch/ex38/access](#)
- **ВНИМАНИЕ.** Мощный пример получения Load Factor через Reflection
- Пример. Реализация функции rehash() [lesson\\_ch17/ch/ex39/access](#)
- **ВНИМАНИЕ.** Мощный пример реализации функции rehash() и алгоритма ПРОСТЫХ ЧИСЕЛ

## Сортировка и поиск в списках

- Пример. реализация сортировки с компаратором List [lesson\\_ch17/ch/ex40/access](#)
- Пример. сортировка с компаратором Set для Map [lesson\\_ch17/ch/ex41/access](#)
- Пример. сортировка с компаратором List по алфавиту (toLowerCase) [lesson\\_ch17/ch/ex40/access](#)
- **ВНИМАНИЕ.** Мощный пример.

## Внутренний класс метода

- Пример. реализация на внутреннем классе [lesson\\_ch18/ch/ex01/local](#)

```
public static void app(String[] args) {  
    // String regex = "(\\w+\\W+)+zip"; // выбранные  
    String regex = ".+"; // все файлы  
    System.out.println("\nDir with Internal Class:");  
    DirList2.getDir(new String[]{regex}); // вместо args[]  
}
```

- Пример. реализация на методе с внутренним классом
- **ВНИМАНИЕ.** args[0] из внешнего класса используемый в внутреннем классе метода должен быть FINAL

```
public class DirList3 {  
    public static void getDir(String[] args) {  
        File path = new File("."); // путь до каталога  
        String[] list; // массив для хранения  
        if (args.length == 0) {  
            list = path.list(); // получить список каталогов  
        } else {  
            list = path.list(new FilenameFilter() { // работает только с final arg  
                private Pattern pattern = Pattern.compile(args[0]);  
                @Override  
                public boolean accept(File dir, String name) {  
                    return pattern.matcher(name).matches();  
                } // искать каталоги по pattern  
            });  
        } //if_else  
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER); // сортировать каталог  
        for (String s : list) {  
            System.out.println(s);  
        }  
    }  
}
```

## Mapped Memory File

- **ВНИМАНИЕ.** удаление файла который был размещен Mapped Memory
  - открыть FileChannel отдельным дескриптором
  - закрыть FileChannel
  - присвоить mbb = null буферу памяти
  - вызвать System.gc()
  - удалить файл
- ```
public static void check() {
    String fileName = "./src/ch18/ex25/codec/test.dat";
    try {

        System.out.println("File writing started:");
        FileChannel fc = new RandomAccessFile(fileName, "rw").getChannel();
        MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE, 0, length);
        System.out.println("file length:"+length);
        for (int i = 0; i < length ; i++) {
            mbb.put((byte) 'x');
        }
        System.out.println("File written");
        System.out.print("File data:");
        for (int i = length/2; i < length/2+10; i++) {
            System.out.print((char) mbb.get(i)+" ");
        }
        System.out.println();
    // ВНИМАНИЕ ОСВОБОЖДЕНИЕ mapped memory file
        fc.close();
        mbb=null;
        System.gc();

    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    new File(fileName).delete();
}
```

## Визуальные потоки Добавление объекта в окно рисования JFrame

- Визуальные потоки
  - добавление задач в объекты Swing
  - запуск локальных фоновых задач внутри объектов Swing
  - repaint() асинхронный метод, можно вызывать без ограничений
- **ВНИМАНИЕ.** repaint() асинхронный метод, только устанавливает флаг запрос
- **ВНИМАНИЕ.** repaint() при переопределении ОБЯЗАТЕЛЬНО вызывать в первой строке paintComponent()  

```
super.paintComponent(g);
```
- **ВНИМАНИЕ.** При ДОБАВЛЕНИИ объекта в окно рисования, все надо затирать и рисовать заново
- Способ 1. Создать МАССИВ для хранения объектов, в paintComponent() рисовать из массива при
- Способ 2. Создать IMAGE и дорисовать объект на существующую картинку, Image потом выводить

*Hints*

## Поточное программирование

- Использование совместных ресурсов составляет Основную проблему при работе в несколько потоков
  - Java поддерживает механизмы работы с **ресурсами на уровне библиотек ???**

## Нисходящее преобразование

- Суть нисходящего преобразования **????**
  - ВНИМАНИЕ.** При обращении к полям и методам потомка, которых нет у предка вылезет ошибка

```
Wind wind = (Wind) new Instrument(); // down casting
```

## Интерфейсы и шаблон Фабрика

- Шаблон фабрика для чего нужен осталось неясным **???**
- Шаблон фабрика работает так на примере Game [lesson\\_ch09/ch/ex18/gamer](#)

## Вложенные классы

- Статический внутренний класс называется вложенный класс [lesson\\_ch10/ch/ex16/ex18/ex19](#)
  - для создания объекта не нужен объект внешнего класса
  -
- Отличие от внутреннего класса
  - внутренний класс имеет область видимости только в пределах внешнего класса **???**
  - вложенный класс имеет область видимости не ограниченную внешним классом **???**

## Добавление групп элементов

- Collection и Arrays есть методы добавления группы элементов [lesson\\_ch11/ch/ex4/local/include](#)
- ВНИМАНИЕ.** Arrays.asList() может некорректно работать с потомками **???** не подтверждается **???**
  - Collections.addAll() предпочтительно

## Finally и операторы continue и break

- Finally также выполняется до перехвата управления continue; и break;
- Вместе с labels он позволяет заменить goto **???**

## Анализ ограничений

- Анализ ограничений Bounds и WildCard <? super T>
- упражнение 28. стр.554 **???**

## Generics vs Unbounded Mask

- Generics vs Unbounded Mask когда и как различаются [lesson\\_ch15/ch/ex29/include](#)
- ВНИМАНИЕ ПРОВЕРИТЬ РАЗЛИЧИЕ ПОХОЖЕ АБЗАЦ УСТАРЕЛ ????**

## Latent и Универсальные Адаптеры

- Универсальные Адаптеры [lesson\\_ch15/ch/ex42/local](#)
- Теория. В общем все довольно сложно и надо будет разобраться Pattern Strategy **????**
- Пример. реализация математики на разных классах Number [lesson\\_ch15/ch/ex42/local](#)
- Пример. реализация на двух разных Custom классах [lesson\\_ch15/ch/ex42/access](#)

## *Нерешенные вопросы*

### **Реализация SlowMap на EntrySet и упрощенного MapEntry**

- SlowMap на базе своего EntrySet и упрощенного MapEntry **??? РАЗОБРАТЬСЯ** [lesson\\_ch17/ch/ex16/local](#)
- **ВНИМАНИЕ.** Похоже все методы поиска и удаления работают через EntrySet.Iterator<>
- Есть два способа добраться до функции keyset().removeAll()
  - реализовать свой keyset() что в принципе неверно, так как это заглушка
  - реализовать свой EntrySet это ВЕРНОЕ решение официальное
- Реализация EntrySet
  - делается наследованием класса AbstractSet<Map.Entry<K,V>>
  - реализуются два метода iterator() и size() по умолчанию
  - внутри метода iterator() при создании класса new Iterator реализуется метод remove()
- **ВНИМАНИЕ.** именно метод newIterator().remove() отрабатывает keyset().remove(), keyset().removeAll()
- 

### **Удержание ссылок**

- Удержание ссылок [lesson\\_ch17/ch/ex42/remote](#)
  - WeakReference используется для реализации «канонического отображения» ???

### **Redirect Standard Stream**

- Перенаправление Standard Stream
  - setIn(inputStream)
  - setOut(PrintStream)
  - setErr(PrintStream)
- **ВНИМАНИЕ.** Как происходит разыменование после SetIn() SetOut() ???
- 

### **Долговременное хранение**

- Долговременное хранение [lesson\\_ch18/ch/ex30/codec](#)
  - это хранение объектов данных на диске для последующего восстановления состояния программы
- Правила работы сохранения объектов
  - не использовать код чтения классов **объектов <????>** стр.804

### **Использование Executor**

- Объект ExecutorService
  - Executor с жизненным **циклом службы** ???
- 
-

**Примеры реализации****Приложение D**

- Реализация работы с объектами типа <Class>
- Пример.

```

public class Individual {
    private static long count;
    protected static final long id = count++;
    protected String name;
    public Individual() {
    }
    public Individual(String name) {
        this.name = name;
    }
}
// Pets =====
• public class Pet extends Individual {
    public Pet() {
        super();
    }
    public Pet(String name) {
        super(name);
    }
    @Override
    public String toString() {
        return "<" +this.getClass().getSimpleName() +">";
    }
}
// Cats =====
• public class Cat extends Pet {
    public Cat() {
        super();
    }
    public Cat(String name) {
        super(name);
    }
}
• public class Manx extends Cat {
    public Manx() {
        super();
    }
    public Manx(String name) {
        super(name);
    }
}
• public class Cymr extends Cat {
    public Cymr() {
        super();
    }
    public Cymr(String name) {
        super(name);
    }
}

```

lesson\_ch14/ch/ex11/value

## Примеры реализации

- Генератор объектов

lesson\_ch14/ch/ex11/value

```
// Pet Maker =====
public abstract class PetMake {
    private Random rnd = new Random();
// ВНИМАНИЕ абстрактный метод выдает СПИСОК типов генерится в потомке
    public abstract List<Class<? extends Pet>> types(); // абстрактный метод выдает тип

    public Pet randomPet() {
        int n = rnd.nextInt(types().size()); // выдать индекс по размеру списка
        try {
            return types().get(n).newInstance(); // по списку выдать случайный объект
        } catch (Exception e) {
            System.out.println("incompatible type:" + types().get(n).getSimpleName());
            throw new RuntimeException(); // переправить в RuntimeException
        }
    }
    public Pet[] getArray(int size) { // генерация массива нужного размера
        Pet[] pets = new Pet[size];
        for (int i = 0; i < size; i++) {
            pets[i] = randomPet();
        }
        return pets;
    }
    public ArrayList<Pet> getList(int size) { // генерация списка нужного размера
        ArrayList<Pet> list = new ArrayList<>();
        Collections.addAll(list, getArray(size));
        return list;
    }
}
// Pet Type Gen =====
public class PetNameL extends PetMake {
// исходный final массив смежных типов классов производных от Pet
    private static final Class<? extends Pet> [] typeArray = new Class[] {
        Pet.class, Dog.class, Cat.class, Rodent.class,
        Pug.class, Mutt.class, EgCat.class, Manx.class, Cymr.class,
        Rat.class, Mouse.class, Hamster.class
    };

    public static final List<Class<? extends Pet>> alltypes =
        Collections.unmodifiableList(Arrays.asList(typeArray)); // генерим полный список

    private static final int iStart = alltypes.indexOf(Pug.class); // часть списка
    private static final int iEnd = alltypes.size();
// types часть списка только животные не включая виды, классы одного уровня
    private static final List<Class<? extends Pet>> types = alltypes.subList(iStart, iEnd);

    @Override
    public List<Class<? extends Pet>> types() {
        return types;
    }
}

// Pet Gen =====
public class Pets {
    public static final PetMake pets = new PetNameL(); // заряжаем потомка в тело предка

    public static Pet randomPet() {
        return pets.randomPet();
    }
    public static Pet[] getArray(int size) { // получить массив объектов shell
        return pets.getArray(size);
    }
    public static List<Pet> getList(int size) { // получить список объектов shell
        return pets.getList(size);
    }
}
```

## Примеры реализации

```
}

• Счетчик объектов. Улучшенная версия          lesson_ch14/ch/ex11/value
// полностью работает как LinkedHashMap<Class Pet, Integer>
public class PetCount2 extends LinkedHashMap<Class<? extends Pet>, Integer> {
    public PetCount2() {
        // создает карту со всеми типами из PetNameL.alltypes и значением 0 для всех записей
        // ВНИМАНИЕ. Использует класс пользователя MapData !!!
        super(MapData.map(PetNameL.alltypes, 0));
    }
    private void count(Pet pet) {                      // так класс наследует
        for (Map.Entry<Class<? extends Pet>, Integer> entry : entrySet()) {
            if (entry.getKey().isInstance(pet)) {
                put(entry.getKey(), entry.getValue() + 1);
            }
        }
    }
    @Override
    public String toString () {
        StringBuilder sb = new StringBuilder("{");
        for (Map.Entry<Class<? extends Pet>, Integer> entry : entrySet()) {
            sb.append(entry.getKey().getSimpleName());
            sb.append(":");
            sb.append(entry.getValue());
            sb.append(",");
        }
        sb.delete(sb.length() - 1, sb.length());
        sb.append("}");
        return sb.toString();
    }
    public void countPet(List<Pet> pets) {
        for (Pet pet : pets) {
            count(pet);
        }
    }
    public void countPet(PetMake pm) {
        for (Pet pet : pm.getArray(20)) {
            count(pet);
        }
    }
}
• Управляющая программа
public class Unit {
    public static void app() throws Exception {
        System.out.println("PetCount2:");
        PetCount2 pc2 = new PetCount2();
        pc2.countPet( Pets.pets);      // LinkedHashMap <String, Integer>
        System.out.println(pc2);
    }
}
```

## Примеры реализации

[lesson\\_ch14/ch/ex11/value](#)

- Счетчик объектов Продвинутая версия

```
public class TypeCount extends HashMap<Class<?>, Integer> {
    private Class<?> baseType;
    public TypeCount(Class<?> baseType) { // назначается базовый класс
        this.baseType = baseType;
    }
    private void countClass(Class<?> type) {
        Integer q = get(type);
        if (q == null) {
            put(type, 1);
        } else {
            put(type, q + 1);
        }
        Class<?> superClass = type.getSuperclass();
    }
    // рекурсивный вызов всей иерархии классов
    if (superClass != null && baseType.isAssignableFrom(superClass)) {
        countClass(superClass);
    }
}
public void count(Object object) {
    Class<?> type = object.getClass(); // получить класс от входного объекта
}
// сравнение является ли класс type потомком класса <baseType> n = new <type>
if (!baseType.isAssignableFrom(type)) {
    throw new RuntimeException("catch: wrong type :" + type.getSimpleName() +
        " must be subclass of:" + baseType.getSimpleName());
}
countClass(type); // посчитать
}
@Override
public String toString() {
    StringBuilder sb = new StringBuilder("{");
    for (Map.Entry<Class<?>, Integer> entry : entrySet()) {
        sb.append(entry.getKey().getSimpleName());
        sb.append(":");
        sb.append(entry.getValue());
        sb.append(",");
    }
    sb.delete(sb.length() - 1, sb.length());
    sb.append("}");
    return sb.toString();
}
```

- Управляющая программа

```
public class Value {
    public static void app() throws Exception {
        System.out.println("\n====VALUE====");
        System.out.println("\nClass isAssignableFrom() Check\n");

        List<Pet> pList = Pets.getList(25);
        System.out.println("PetCount2:");
        PetCount2 pc2 = new PetCount2();
        pc2.countPet( pList); // LinkedHashMap <String, Integer>
        System.out.println(pc2);

        System.out.println("TypeCount:");
        TypeCount tp = new TypeCount(Pet.class); // создать объект счетчика
        for (Pet pet : pList) {
            tp.count(pet);
        }
        System.out.println(tp);
    }
}
```

*Примеры реализации*

- Внутри класса

```
public static class NullPerson extends Person implements Null { // встроенный класс
    private NullPerson() { // нельзя создать объект класса
        super("None", "None", "None");
    }
    @Override
    public String toString() {
        return String.format("Person: <NULL>");
    }
}
public static final Person NULL = new NullPerson(); // статический экземпляр
```

- Снаружи

о это не SinglTon, так как идет расширение класса экземпляр NULL класс AirFilter один из многих

- `public class AirFilterNull extends AirFilter {
 public static AirFilter NULL = new AirFilterNull();
 private AirFilterNull() {
 super();
 }
 @Override
 public String toString() {
 return NULL.getClass().getSimpleName() + ".NULL";
 }
}`

- Dynamic Proxy

```
public class NullRobotProxy implements InvocationHandler {
    private String nullName; // имя для пустого объекта робот
    private Robot proxyObj = new NRobot(); // встроенный класс
    private class NRobot implements Null, Robot { // оба интерфейса проприетарные
        public String name() {
            return nullName;
        }
        public String model() {
            return nullName;
        }
        public List<Operate> operate() {
            return Collections.emptyList(); // возвращает пустой список
        }
    } // class NRobot finished
    public NullRobotProxy(Class <? extends Robot> type) { // производными от Robot
        nullName = type.getSimpleName() + " NULLRobot"; // от имени класса объекта
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        return method.invoke(proxyObj, args); // вызвать оригинальный метод
    }
}
```

- Инициализация >> NullRobot.getNULL(SnowRobot.class)

- `public class NullRobot {
 public static Robot getNULL(Class<?extends Robot> type) { // единственный метод
 return (Robot) Proxy.newProxyInstance(
 NullRobot.class.getClassLoader(),
 new Class[] { Null.class, Robot.class },
 new NullRobotProxy(type)
 );
 }
}`