УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационной безопасности

Кафедра инфокоммуникационных технологий

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ ИНФОКОММУНИКАЦИОННЫХ СИСТЕМ
Часть 2

Практическая работа 4
Универсальные типы. Коллекции. Потоки

Минск 2023

Макейчик Е.Г.

Содержание

Практическая работа 4
Универсальные типы. Коллекции. Потоки

Универсальные типы (дженерики)

По сути, универсальные типы должны были бы называться параметризованными типами. Они очень важны, поскольку позволяют создавать классы, интерфейсы и методы, для которых типы обрабатываемых данных передаются в качестве параметра. Классы, интерфейсы или методы, которые обрабатывают типы, передаваемые посредством параметров, называются универсальными (дженериками), например, можно говорить об универсальном классе или универсальном методе.

Главное преимущество универсального кода состоит в том, что он автоматически настраивается на работу с нужным типом данных. Многие алгоритмы выполняются одинаково, независимо от того, к информации какого типа данных они должны быть применены. Например, быстрая сортировка не зависит от типа данных, в качестве которого можно использовать Integer, String, Object и даже Thread. Используя универсальные типы, можно единожды реализовать алгоритм, а затем без труда применять его к любому типу данных.

Язык Java всегда давал возможность создавать классы, интерфейсы и методы, пригодные для обработки любых типов данных. Это достигалось за счет использования класса Object. Поскольку Object является суперклассом для всех остальных классов, ссылка на Object может выступать в качестве ссылки на любой другой объект. Таким образом, до появления универсального кода для выполнения действий с любыми типами данных в классах, объектах и методах использовалась ссылка типа Object. При этом возникала серьезная проблема, связанная с необходимостью явно преобразовывать Object в другой тип. Подобное преобразование часто становилось источником ошибок. Универсальные типы повышают уровень безопасности при работе с данными, поскольку при этом все преобразования типов происходят неявно. Таким образом, универсальные типы повышают пригодность кода к повторному использованию.

Ниже представлен синтаксис объявления универсального класса и синтаксис объявления ссылки на универсальный класс.

```
class имя_класса<параметры_типа> { // ...

имя_класса<передаваемые_тип имя_переменной =
new имя_класса<передаваемые_типы>(передаваемые_значения);
```

В коде ниже приведен простой пример использования универсальных типов.

```java
// Здесь T - это параметр, который заменяется реальным именем типа при создании объекта Gen
public class Gen<T> {
    private T ob; // Объявление объекта типа T
    // Конструктору передается ссылка на объект типа T
    public Gen(T o) {
        ob = o;
    }
    // Метод возвращает объект ob
    public T getob() {
        return ob;
    }
    // Отображение типа T
    public void showType() {
        System.out.println("Type of T is "+ ob.getClass().getName());
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Создание ссылки на объект типа Gen<Integer>
        Gen<Integer> iOb;
        // Создание объекта Gen<Integer> и присваивание ссылки на него переменной iOb
        // При помещении значения 88 в состав объекта Integer используется автоупаковка
        iOb = new Gen<Integer>( o: 88);
        // Отображение типа данных, используемого iOb
        iOb.showType();
        // Получение значения iOb. Приведение типов не требуется
        int v = iOb.getob();
        System.out.println("value: " + v);
        System.out.println();
        // Создание ссылки на объект типа Gen<String>
        Gen<String> strOb = new Gen<String>( o: "Generics Test");
        // Отображение типа данных, используемого strOb
        strOb.showType();
        // Получение значения strOb. Приведение типов также не требуется
        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}
```

```
Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test
```

Для объявления Gen используется следующая строка кода.

```
public class Gen<T> {
```

где T – это имя параметра типа, т.е. параметра, с помощью которого задается тип данных. Данное имя впоследствии будет заменено реальным именем типа, которое передается Gen при создании объекта. Таким образом, T используется в объекте Gen там, где необходим параметр типа. Объявляемый параметр типа указывается в угловых сколках. Поскольку в Gen используется параметр типа, Gen является универсальным классом (дженериком).

Вместо имени T может быть использован любой допустимый идентификатор, но традиционно разработчики применяют имя T. Кроме того, настоятельно рекомендуется, чтобы параметр типа состоял из одной прописной буквы. Если при объявлении класса надо задать несколько параметров типа, то кроме T часто используют имена V и E.

В следующем выражении T используется для объявления объекта ob.

```
private T ob;
```

Как было сказано ранее, идентификатор T предназначен для того, чтобы быть замещенным реальным типом при создании объекта Gen. Таким образом, в качестве типа объекта ob будет принят тип, переданный посредством T. При создании объекта будет указан тип String, и объект ob будет также принадлежать типу String.

Рассмотрим создание конструктора класса Gen.

```
public Gen (T o) {
ob = o
}
```

Параметр o данного конструктора принадлежит T. Это означает, что реальный тип o определяется типом, переданным посредством T при создании объекта Gen. Поскольку параметр o и переменная ob принадлежат типу T, то после замены T они также будут принадлежать одному и тому же реальному типу.

Параметр типа T можно также использовать для того, чтобы задать тип значения, возвращаемого методом. Примером может служить метод getob(), код которого показан ниже.

```
public T getob() {
return ob;
}
```

Поскольку типом переменной ob является T, она совместима с типом, возвращаемым методом getob().

Метод showType() отображает тип T. Эта задача решается путем вызова метода getName() объекта Class, полученного в результате вызова метода getClass() объекта ob. В классе Object определен метод getClass(). Таким образом, данный метод является членом каждого класса. Он возвращает объект Class, соответствующий типу текущего объекта. Класс Class принадлежит пакету java.lang и инкапсулирует информацию о классе. В нем определено несколько методов, которые позволяют в процессе выполнения программы получать сведения о классе. К их числу относится метод getName(), возвращающий строковое представление имени класса.

Класс Main демонстрирует работу универсального класса Gen. В первую очередь в нем создается вариант объекта Gen для целых чисел.

```
Gen<Integer> iOb;
```

Тип Integer указывается после имени Gen в угловых скобках. В данном случае Integer – это передаваемый тип, который заменяет в классе Gen параметр типа T. Таким образом создается вариант Gen, в котором ссылки на T преобразуются в ссылки на Integer. В объекте, созданном при выполнении приведенного выражения, переменная ob и возвращаемое значение метода getob() будут принадлежать типу Integer.

На самом деле компилятор Java реально не создает различные версии Gen или другого универсального класса. В принципе удобно было бы считать, что это происходит, но на самом деле все обстоит по-другому. Вместо создан разных версий компилятор удаляет всю информацию об универсальном типе и использует вместо нее приведение типа. В результате полученный объект ведет себя так, как будто в программе была создана конкретная версия класса Gen. Таким образом, в программе реально существует лишь одна версия Gen. Процесс удаления информации об универсальном типе называется стиранием (erasure).

В следующей строке кода переменной iOb присваивается ссылка на экземпляр Integer-варианта класса Gen.

```
iOb = new Gen<Integer>(88);
```

При вызове конструктора Gen задается передаваемый тип Integer. Это необходимо, поскольку тип объекта, на который указывает ссылка (в данном случае это iOb), должен соответствовать Gen<Integer>.

При его выполнении осуществляется автоупаковка целочисленного значения 88 в объект Integer. Дело в том, что конструктору Gen<Integer> должен передаваться параметр типа Integer. При необходимости преобразование типов может быть выполнено явно, как в следующем примере.

```
iOb = new Gen<Integer>(new Integer(88));
```

Однако в данном случае длинная строка кода не дает никаких преимуществ по сравнению с предыдущим, более компактным выражением.

Если тип ссылки, возвращаемой оператором new, будет отличаться от Gen<Integer>, возникнет ошибка компиляции. Например, сообщение о такой ошибке будет сгенерировано при попытке скомпилировать выражение.

```
iOb = new Gen<Double>(88.o); // Ошибка!
```

Поскольку переменная iOb принадлежит типу Gen<Integer>, ее нельзя использовать для хранения ссылки, например, на объект Gen<Double>. Возможность проверки на соответствие типов – одно из основных преимуществ универсальных типов.

Далее в программе отображается тип переменной ob, принадлежащей объекту iOb (в данном случае это тип Integer). Для получения значения ob используется следующее выражение.

```
int v = iOb.getob();
```

Поскольку метод getob() возвращает значение типа T, которое заменяется Integer, возвращаемым типом getob() будет Integer. Это значение перед присвоением переменной v (типа int) будет подвергнуто автораспаковке.

И наконец, в классе Main объявляется объект типа Gen<String>.

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Поскольку параметр типа – это String, данный тип заменяет T в составе Gen. Таким способом создается String-вариант Gen, работу которого демонстрируют оставшиеся строки кода.

При определении экземпляра универсального класса передаваемый тип, который заменяет параметр типа, должен быть именем класса. Для этой цели нельзя использовать простой тип, например, int или char. В случае класса Gen можно передать в качестве T любой класс, но не простой тип. Другими словами, следующее выражение недопустимо.

```
Gen<int> strOb = new Gen<int>(53); // Ошибка.  Простой тип использовать нельзя
```

Понятно, что запрет на использование простых типов не является серьезным ограничением, так как всегда можно использовать класс оболочки, инкапсулировав в нем требуемое значение Поддержка автоупаковки и автораспаковки еще больше упрощает данный подход.

Необходимо заметить, что ссылка на конкретный вариант одного универсального типа несовместима с объектом другого универсального типа. Например, если бы в рассмотренной ранее программе присутствовала приведенная ниже строка кода, компилятор сгенерировал бы сообщение об ошибке и не сформировал файл класса.

```
iOb = strOb; // Ошибка!
```

Несмотря на то что и iOb и strOb принадлежат типу Gen<T>, они являются ссылками на различные типы. Такое несоответствие возникает вследствие различия передаваемых типов. Эта особенность универсальных типов предотвращает ошибки при написании программ.

Универсальный класс с двумя параметрами типа

В универсальном классе можно задать несколько параметров типа. В этом случае параметры разделяются запятыми. Ниже приведен класс TwoGen (модификация класса Gen) и в нем определены два параметра типа.

```java
// Простой универсальный класс с двумя параметрами типа: T и V
public class TwoGen<T, V> {
    private T ob1;
    private V ob2;
    // Конструктору класса передаются ссылки на объекты типов T и V
    public TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // Отображение типов T и V
    public void showTypes() {
        System.out.println("Type of T is "+ ob1.getClass().getName());
        System.out.println("Type of V is "+ ob2.getClass().getName());
    }
    public T getob1() {
        return ob1;
    }
    public V getob2() {
        return ob2;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // В данном случае параметр T заменяется типом Integer, а параметр V - типом String
        TwoGen<Integer, String> tgObj =
                new TwoGen<Integer, String>( o1: 88,  o2: "Generics");
        // Отображение типов
        tgObj.showTypes();
        // Получение и отображение переменных
        int v = tgObj.getob1();
        System.out.println("value: " + v);
        String str = tgObj.getob2();
        System.out.println("value: " + str);
    }
}
```

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Здесь определяются два параметра типа: T и V; они разделяются запятыми. Поскольку в классе используются два параметра типа, при создании объекта надо указывать два передаваемых типа.

```
TwoGen<Integer, String> tgObj =
        new TwoGen<Integer, String>(88, "Generics");
```

В данном случае Integer заменяет параметр типа T, а String заменяет V. Хотя в данном примере передаваемые типы различаются, в принципе они могут совпадать. Например, следующая строка кода допустима.

```
TwoGen<String, String> x =
        new TwoGen<String, String>("A", "B");
```

В данном случае T и V заменяются типом String. Конечно, если передаваемые типы всегда совпадают, определять два параметра типа нет никакой необходимости.

Ограниченные типы

В предыдущих примерах параметры типа могли заменяться любым классом. Такое решение подходит для многих целей, но иногда полезно ограничить допустимый набор передаваемых типов. Предположим, необходимо создать универсальный класс, который хранил бы числовое значение и мог выполнять различные математические функции, например, вычислять обратное значение или извлекать дробную часть. Предположим, что нужно использовать данный класс для работы с любыми числовыми типами: целочисленными и с плавающей точкой.

Для подобных ситуаций в языке Java предусмотрены ограниченные типы. При объявлении параметра типа можно указать так называемую верхнюю границу, т.е. задать суперкласс, который должны наследовать все передаваемые типы. С этой целью было введено выражение extends, определяющее параметр типа так, как показано ниже.

```
<T extends суперкласс>
```

С помощью данного выражения компилятору предоставляется информация о том, что T может быть заменен только суперклассом или его подклассами. Таким образом, суперкласс определяет верхнюю границу в иерархии классов в Java.

В коде ниже приведен пример использования ограниченного параметра.

```java
public class NumericFns<T extends Number> {
    private T num;
    public NumericFns(T n) {
        num = n;
    }
    public double reciprocal() {
        return 1 / num.doubleValue();
    }
    public double fraction() {
        return num.doubleValue() - num.intValue();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Применение Integer допустимо, поскольку данный класс является подклассом Number
        NumericFns<Integer> iOb =new NumericFns<Integer>( n: 5);
        System.out.println("Reciprocal of iOb is "+ iOb.reciprocal());
        System.out.println("Fractional component of iOb is "+ iOb.fraction());
        System.out.println();
        // Применение Double также допустимо
        NumericFns<Double> dOb =new NumericFns<Double>( n: 5.25);
        System.out.println("Reciprocal of dOb is " + dOb.reciprocal());
        System.out.println("Fractional component of dOb is "+ dOb.fraction());
        // Следующая строка кода не будет компилироваться, поскольку String не является подклассом Number
        // NumericFns<String> strOb = new NumericFns<String>("Error");
    }
}
```

```
Reciprocal of iOb is 0.2
Fractional component of iOb is 0.0

Reciprocal of dOb is 0.19047619047619047
Fractional component of dOb is 0.25
```

Теперь для объявления NumericFns используется следующая строка кода.

```
public class NumericFns<T extends Number> {
```

Поскольку тип T теперь ограничен классом Number, компилятор Java знает, что все объекты типа T содержат метод doubleValue(), а также другие методы определенные в Number. Это само по себе является огромным преимуществом. Данный механизм еще предотвращает создание объектов NumericFns типов, отличных от числовых. Например, если попытается удалить комментарии из строк, расположенных в конце программы, а затем попытаетесь повторно скомпилировать код, будет выведено сообщение об ошибке, поскольку String не является подклассом Number.

Задание

Необходимо написать параметризированный класс с двумя параметрами, ограниченными классом Number и разработать метод вычисления суммы двух чисел, любых типов, метод сравнивающий поэлементно два массива разных типов, метод определяющий наибольший и наименьший элементы в массивах.

Использование групповых параметров

Рассмотренные выше варианты универсальных типов удобны в работе, но в ряде случаев приходится отдавать предпочтение другим конструкциям. Предположим, что необходимо реализовать метод absEqual(), который возвращал бы значение true в случае, если два объекта NumericFns (этот класс был рассмотрен ранее) содержат одинаковые значения. Предположим также, что необходимо, чтобы этот метод работал с любыми типами данных, которые могут храниться в сравниваемых объектах. Например, если один объект содержит значение 1,25 типа Double, а второй – значение –1,25 типа Float, метод absEqual() должен возвращать значение true. Один из способов реализации absEqual() состоит в том, чтобы передавать методу параметр типа NumericFns, а затем сравнивать его абсолютное значение с абсолютным значением текущего объекта и возвращать true, если эти значения совпадают. Например, вызов absEqual() может выглядеть следующим образом.

```
NumericFns<Double> dOb = new NumericFns<Double>(1.25);
NumericFns<Float> fOb = new NumericFns<Float>(-1.25);
if (dOb.absEqual(fOb) )
  System.out.println("Absolute values are the same.");
else
  System.out.println("Absolute values differ.");
```

На первый взгляд кажется, что при работе с absEqual() не возникают никакие проблемы. Однако это не так. Неприятности начнутся сразу же, как только попытается объявить параметр типа NumericFns. Подходящим кажется решение наподобие следующего, где в качестве параметра типа задается T.

```
// Программа работает некорректно!
// Определение равенства абсолютных значений двух объектов
public boolean absEqual(NumericFns<T> ob) {
if (Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue())
  return true;
else
  return false;
}
```

В данном случае для определения абсолютного значения каждого числа используется стандартный метод Math.abs(). Полученные значения сравниваются. Проблема состоит в том, что описанный подход будет действовать только тогда, когда объект NumericFns, передаваемый в качестве параметра, имеет тот же тип, что и текущий объект. Например, если текущий объект принадлежит типу NumericFns<Integer>, параметр ob также должен быть типа NumericFns<Integer>. Сравнить текущий объект с объектом типа NumericFns<Double> невозможно. Таким образом, решение не является универсальным.

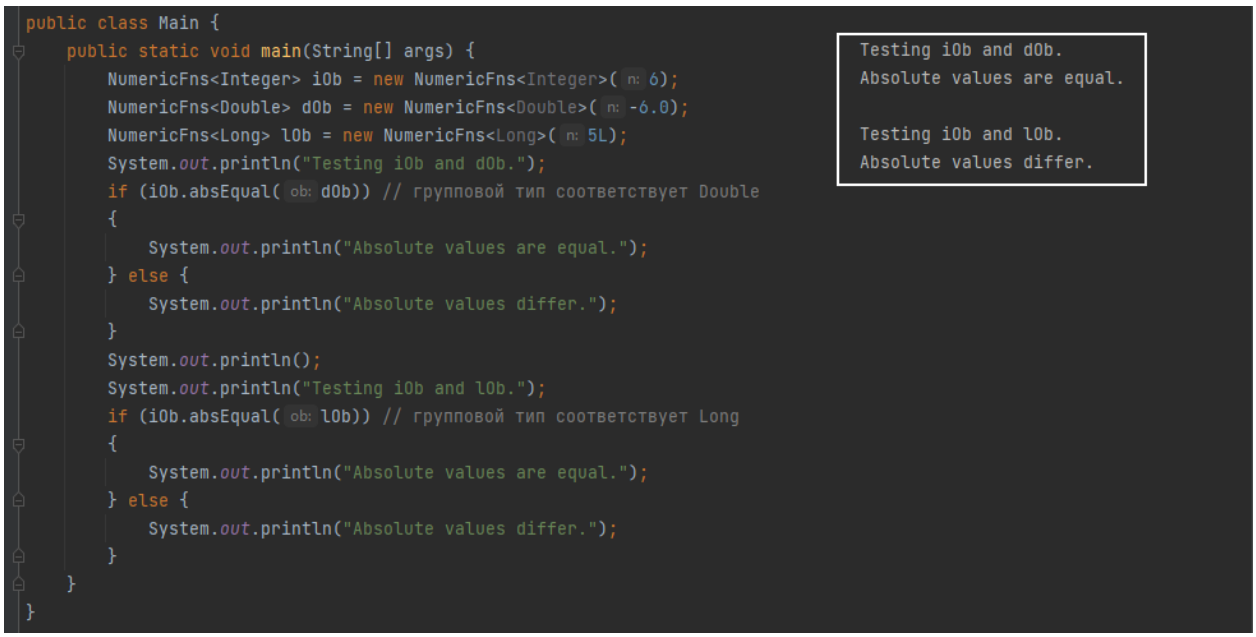
Для того чтобы создать метод absEqual(), применимый во всех случаях, необходимо использовать еще одно средство, связанное с универсальными типами, – групповой параметр. В качестве такого параметра используется символ ?, который представляет неизвестный тип.

Используя данное средство, метод absEqual() можно переписать так.

```
public boolean absEqual(NumericFns<?> ob) {
if (Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()))
  return true;
else
  return false;
}
```

Здесь выражение NumericFns<?> соответствует любому типу NumericFns и позволяет сравнивать абсолютные значения двух произвольных объектов NumericFns.

```java
// Использование группового параметра
public class NumericFns<T extends Number> {
    private T num;
    public NumericFns(T n) {
        num = n;
    }
    public double reciprocal() {
        return 1 / num.doubleValue();
    }
    public double fraction() {
        return num.doubleValue() - num.intValue();
    }
    public boolean absEqual(NumericFns<?> ob) {
        if (Math.abs(num.doubleValue())
                == Math.abs(ob.num.doubleValue())) {
            return true;
        } else {
            return false;
        }
    }
}
```



```
public class Main {
    public static void main(String[] args) {
        NumericFns<Integer> iOb = new NumericFns<Integer>( n: 6);
        NumericFns<Double> dOb = new NumericFns<Double>( n: -6.0);
        NumericFns<Long> lOb = new NumericFns<Long>( n: 5L);
        System.out.println("Testing iOb and dOb.");
        if (iOb.absEqual( ob: dOb)) // групповой тип соответствует Double
        {
            System.out.println("Absolute values are equal.");
        } else {
            System.out.println("Absolute values differ.");
        }
        System.out.println();
        System.out.println("Testing iOb and lOb.");
        if (iOb.absEqual( ob: lOb)) // групповой тип соответствует Long
        {
            System.out.println("Absolute values are equal.");
        } else {
            System.out.println("Absolute values differ.");
        }
    }
}
```

```
Testing iOb and dOb.
Absolute values are equal.

Testing iOb and lOb.
Absolute values differ.
```

```
if (iOb.absEqual(dOb))
if (iOb.absEqual(lOb))
```

В первом выражении iOb – это объект типа NumericFns<Integer>, а dOb – объект типа NumericFns<Double>. Благодаря использованию группового параметра становится возможным передать dOb методу absEqual(). Подобным образом формируется и другой вызов, в котором методу передается объект типа NumericFns<Long>.

Важно понимать, что групповые параметры не влияют на то, какой тип объекта NumericFns может быть создан. Этим управляет выражение extends в объявлении NumericFns. Групповой параметр лишь указывает на соответствие любому объекту NumericFns.

Универсальные методы

Как было показано в предыдущих примерах, методы в составе универсальных классов могут использовать параметры типов, заданные для классов. Следовательно, такие методы автоматически становятся универсальными относительно параметров типов. Можно, однако, объявить универсальный метод, использующий один или несколько собственных параметров типов. Более того, такой метод может принадлежать обычному, не универсальному классу.

Ниже приведен код программы, в котором объявлен класс GenericMethodDemo, не являющийся универсальным. В этом классе содержится статический универсальный метод arraysEqual(), который определяет, содержатся ли в двух массивах одинаковые элементы, расположенные в том же порядке. Такой метод можно использовать для сравнения двух массивов одинаковых или совместимых между собой типов.

```
public static <T, V extends T> boolean arraysEqual(T[] x, V[] y) {
```

Параметры типа указаны перед типом значения, возвращаемого методом. Верхней границей V является T, таким образом, V должен принадлежать или тому же типу, что и T, или быть его подклассом. Такая связь гарантирует, что при вызове метода arraysEqual() могут быть указаны только параметры, совместимые друг с другом. Метод arraysEqual() объявлен как static, т.е. его можно вызывать, не создавая предварительно объект. Однако универсальные методы не обязательно должны быть статическими. В этом смысле на них не накладываются ограничения. Теперь рассмотрим обращение к arraysEqual() из тела метода

main(). Для этого используется привычный всем синтаксис, и параметры типа не указываются. Такое решение становится возможным потому, что параметры автоматически распознаются, и типы T и V настраиваются соответствующим образом.

```
public class GenericMethodDemo {
    public static <T, V extends T> boolean arraysEqual(T[] x, V[] y) {
        if (x.length != y.length) {
            return false;
        }
        for (int i = 0; i < x.length; i++) {
            if (!x[i].equals(y[i])) {
                return false; // Массивы различаются
            }
        }
        return true; // Содержимое массивов совпадает
    }
    public static void main(String args[]) {
        Integer nums[] = {1, 2, 3, 4, 5};
        Integer nums2[] = {1, 2, 3, 4, 5};
        Integer nums3[] = {1, 2, 7, 4, 5};
        Integer nums4[] = {1, 2, 7, 4, 5, 6};
        // Параметры типов T и V неявно определяются при вызове метода
        if (arraysEqual( x: nums, y: nums)) {
            System.out.println("nums equals nums");
        }
        if (arraysEqual( x: nums, y: nums2)) {
            System.out.println("nums equals nums2");
        }
        if (arraysEqual( x: nums, y: nums3)) {
            System.out.println("nums equals nums3");
        }
        if (arraysEqual( x: nums, y: nums4)) {
            System.out.println("nums equals nums4");
        }
        // Создание массива типа Double
        Double dvals[] = {1.1, 2.2, 3.3, 4.4, 5.5};
        // Следующая строка не будет скомпилирована, поскольку типы nums и dvals не совпадают
        // if(arraysEqual(nums, dvals))
        // System.out.println("nums equals dvals");
    }
}
```

```
nums equals nums
nums equals nums2
```

if (arraysEqual(nums, nums))

В данном случае базовый тип первого параметра – Integer; этот тип заменяет T. Таким же является базовый тип второго параметра, следовательно, V также заменяется типом Integer. Таким образом, выражение для вызова arraysEqual() составлено корректно и два массива можно сравнить между собой.

Рассмотрим закомментированные строки.

// if (arraysEqual(nums, dvals))
// System.out.println("nums equals dvals");

Если удалить символы комментариев и попытается скомпилировать программу, компилятор отобразит сообщение об ошибке. Причина в том, что верхней границей параметра V является T; этот тип указан после ключевого слова extends. Это означает, что V может либо совпадать с T, либо представлять собой один из его подклассов. В данном случае первый параметр принадлежит типу Integer, им заменяется параметр типа T, однако второй параметр имеет тип Double, который не является подклассом Integer. Таким образом, обращение arraysEqual() становится недопустимым и возникает ошибка на этапе компиляции.

Ниже приведен общий формат определения универсального метода.

<параметры типа> возвращаемый_тип имя_метода (параметры) {

Параметры, как и при вызове обычного метода, разделяются запятыми. Список параметров типа предшествует возвращаемому типу.

Конструктор может быть универсальным, даже если сам класс не является таковым. В коде ниже класс Summation не универсальный, но в нем используется универсальный конструктор.

```
// Использование универсального конструктора
public class Summation {
    private int sum;
    // Универсальный конструктор
    public <T extends Number> Summation(T arg) {
        sum = 0;
        for (int i = 0; i <= arg.intValue(); i++) {
            sum += i;
        }
    }
    public int getSum() {
        return sum;
    }
}
```

```
public class GenConsDemo {
    public static void main(String args[]) {
        Summation ob = new Summation( arg: 4.0);
        System.out.println("Summation of 4.0 is " + ob.getSum());
    }
}
```

```
Summation of 4.0 is 10
```

Класс Summation вычисляет и инкапсулирует сумму всех чисел от 0 до N. Значение N передается конструктору. Поскольку для конструктора Summation() указан параметр типа, ограниченный сверху классом Number, объект Summation может быть создан с использованием любого числового типа, в том числе Integer, Float, Double. Независимо от используемого числового типа, соответствующее значение преобразуется в тип Integer путем вызова intValue() и вычисляется требуемая сумма. Таким образом, совсем не обязательно объявлять класс Summation универсальным; достаточно, если универсальным будет лишь его конструктор.

Универсальные интерфейсы

Наряду с универсальными классами и методами существуют также универсальные интерфейсы. Универсальные интерфейсы определяются точно так же, как и универсальные классы. Их использование иллюстрирует следующий пример. В нем создается интерфейс Containment, который может быть реализован классами, хранящими одно или несколько значений. Кроме того, в нем объявлен метод contains(), который определяет, содержится ли указанное значение в текущем объекте.

```
// Данный интерфейс подразумевает, что реализующий его класс содержит одно или несколько значений
public interface Containment<T> {
    // Метод contains() проверяет, содержится ли некоторый элемент в составе объекта, реализующего Containment
    public boolean contains(T o);
}
```

```
// Реализация интерфейса Containment с использованием массива, предназначенного для хранения значений.
// Любой класс, реализующий универсальный интерфейс, также должен быть универсальным
public class MyClass<T> implements Containment<T> {
    private T[] arrayRef;
    public MyClass(T[] o) { arrayRef = o; }
    public boolean contains(T o) {
        for (T x : arrayRef) {
            if (x.equals(o)) {
                return true;
            }
        }
        return false;
    }
}
```

```
public class GenIFDemo {
    public static void main(String args[]) {
        Integer x[] = {1, 2, 3};
        MyClass<Integer> ob = new MyClass<Integer>( o: x);
        if (ob.contains(2)) {
            System.out.println("2 is in ob");
        } else {
            System.out.println("2 is NOT in ob");
        }
        if (ob.contains(5)) {
            System.out.println("5 is in ob");
        } else {
            System.out.println("5 is NOT in ob");
        }
        // Следующие строки кода недопустимы, так как ob представляет собой вариант Integer
        // реализации интерфейса Containment, а 9.25 – это значение Double
        // if(ob.contains(9.25)) // Illegal!
        // System.out.println("9.25 is in ob");
    }
}
```

```
2 is in ob
5 is NOT in ob
```

В основном элементы этой программы просты для восприятия, однако на некоторых ее особенностях следует остановиться. Прежде всего обратим внимание на то, как определен интерфейс Containment.

```
public interface Containment<T> {
```

Универсальные интерфейсы объявляются так же, как и универсальные классы. В данном случае параметр типа T задает тип включаемого объекта. Интерфейс Containment реализуется классом MyClass. Определение этого класса выглядит следующим образом.

```
public class MyClass<T> implements Containment<T> {
```

Если класс реализует универсальный интерфейс, то он также должен быть универсальным. В нем должен быть объявлен как минимум тот параметр типа, который указан для интерфейса. Например, приведенный ниже вариант объявления класса MyClass недопустим.

```
public class MyClass implements Containment<T> { // Ошибка!
```

В данном случае ошибка заключается в том, что в MyClass не объявлен параметр типа, а это значит, что нет возможности передать параметр типа интерфейсу Containment. При этом идентификатор T неизвестен и компилятор генерирует сообщение об ошибке. Класс, реализующий универсальный интерфейс, может не быть универсальным только в одном случае: если при объявлении класса для интерфейса указывается конкретный тип. Подобный вариант объявления класса приведен ниже.

```
public class MyClass implements Containment<Double> { // oK
```

Один или несколько параметров типа для универсального интерфейса тоже могут быть ограничены. Это позволяет указывать, какие типы данных допустимы для интерфейса. Например, если запретить передачу Containment значений, не являющихся числовыми, используется объявление.

```
public interface Containment<T extends Number> {
```

Теперь любой класс, реализующий Containment, должен передавать интерфейсу значение типа, удовлетворяющее указанным ограничениям. Например, класс MyClass, реализующий рассмотренный интерфейс, должен объявляться следующим образом.

```
public class MyClass<T extends Number> implements Containment<T> {
```

Параметр типа T объявляется в классе MyClass, а затем передается интерфейсу Containment. Поскольку на этот раз интерфейсу Containment требуется тип, расширяющий Number, в классе, реализующем интерфейс (в данном случае это MyClass), должны быть определены соответствующие

ограничения. Если верхняя граница задана в определении класса, то нет необходимости еще раз указывать ее в выражении implements. Более того, если попытаться сделать это, будет выведено сообщение об ошибке. Например, следующее выражение некорректно и не будет компилироваться:

```
// Ошибка!
public class MyClass<T extends Number> implements Containment<T extends Number> {
```

Если параметр типа задан в определении класса, он лишь передается интерфейсу без дальнейшей модификации.

Формат объявления универсального интерфейса выглядит следующим образом

```
public interface имя_интерфейса<параметры_типа> {
```

Здесь параметры типа в составе списка разделяются запятыми. При реализации интерфейса имя класса также должно сопровождаться параметрами типа. Выражение, с помощью которого определяется класс, реализующий интерфейс, показано ниже.

```
class имя_класса<параметры_типа>
implements имя_интерфейса<параметры_типа> {
```

Задание

Необходимо разработать очередь, в которой можно хранить объекты любых типов. Методами get() и set() можно извлечь и добавить объект. Предусмотреть обработку исключений в случае переполнения и пустой очереди.

Ошибки неоднозначности

Появление универсальных типов стало причиной возникновения новых ошибок, связанных с неоднозначностью. Ошибки неоднозначности возникают тогда, когда в результате стирания два, на первый взгляд различающихся универсальных объявления преобразуются в один тип, вызывая тем самым конфликтную ситуацию.

```
// Неоднозначность, вызванная стиранием перегруженных методов
public class MyGenClass<T, V> {
    private T ob1;
    private V ob2;
    // Два следующих метода конфликтуют друг с другом поэтому код не компилируется
    public void set(T o) {
        ob1 = o;
    }
    public void set(V o) {
        ob2 = o;
    }
}
```

В классе MyGenClass объявлены два универсальных типа: T и V и сделана попытка перегрузки метода set(). Перегруженные методы отличаются типами параметров, в данном случае это T и V. Кажется, что ошибки не должны возникать, так как T и V – различные типы. Однако здесь две проблемами, связанными с неоднозначностью. Во-первых, в классе MyGenClass нет никакой гарантии, что T и V действительно будут различаться. Так, например, не является ошибкой создание объекта MyGenClass с помощью выражения.

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В данном случае T и V заменяются String. В результате оба варианта метода set() становятся одинаковыми, что безусловно является ошибкой.

Вторая, более серьезная проблема состоит в том, что в результате стирания оба варианта метода set() преобразуются в следующий вид.

```
void set(Object o) {
```

Таким образом, попытка перегрузки в классе MyGenClass в принципе приводит к неоднозначности определений метода set(). Решением данной проблемы является отказ от перегрузки и использование двух различных имен методов.

Ограничения универсальных типов

Существует ряд ограничений, которые необходимо учитывать при работе с универсальными типами. К ним относятся создание объектов, тип которых является параметром типа, использование статических членов, генерация исключений и работа с массивами.

```
// Экземпляр T создать невозможно
public class Gen<T> {
    private T ob;
    public Gen() {
        ob = new T(); // Illegal!!!
    }
}
```

В данном случае попытка создания экземпляра класса T приводит к ошибке. Причину этого понять несложно. Поскольку при выполнении программы T не существует, компилятор не знает о том, какого типа объект должен быть сформирован. В результате стирания все параметры типа удаляются на этапе компиляции.

Статические члены не могут использовать параметры типа, объявленные в содержащем их классе. Все объявления статических членов в приведенном ниже классе недопустимы.

```
// Экземпляр T создать невозможно
public class Gen<T> {
    // Статическую переменную типа T создать невозможно
    private static T ob;
    // Статический метод не может использовать T
    public static T getOb() {
        return ob;
    }
    // Статический метод не может обращаться к объекту типа T
    public static void showOb() {
        System.out.println(ob);
    }
}
```

Несмотря на наличие описанного выше ограничения, допустимо объявление статических универсальных методов, в которых используются собственные параметры типа. Примеры таких объявлений приводились ранее.

Чтобы при работе с универсальными типами использовать массивы, необходимо учитывать два существенных ограничения. Во-первых, невозможно создать экземпляр массива, базовый тип которого определяется параметром типа. Во-вторых, нельзя создать массив универсальных ссылок, тип которых уточнен. Код ниже демонстрирует оба случая.

```
// Универсальные типы и массивы
public class Gen<T extends Number> {
    private T ob;
    private T vals[]; // Допустимо
    public Gen(T o, T[] nums) {
        ob = o;
        // Следующее выражение недопустимо. Невозможно создать массив типа T
        // vals = new T[10];
        // Следующее выражение корректно
        vals = nums; // Переменной можно присваивать ссылку на существующий массив
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Integer n[] = {1, 2, 3, 4, 5};
        Gen<Integer> iOb = new Gen<Integer>( o: 50,  nums: n);
        // Невозможно создать массив универсальных ссылок
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Ошибка!
        Gen<?> gens[] = new Gen<?>[10]; // Выражение корректно
    }
}
```

Как видно из кода программы, допустимо создать ссылку на массив типа T.

T vals[]; // Допустимо

Однако сам массив T сформировать нельзя. По этой причине приведенная ниже строка закомментирована.

// vals = new T[10]; // Невозможно создать массив типа T

Причина данного ограничения состоит в том, что тип T не существует при выполнении программы, поэтому компилятор не знает, какого типа массив надо в действительности создать.

Однако можно передать ссылку на массив конкретного типа конструктору Gen() при создании объекта, а также присвоить это значение переменной vals. Примером может служить следующая строка.

vals = nums; // Допускается присвоение переменной ссылки на существующий массив

Данное решение корректно, поскольку тип массива, передаваемого Gen, известен. При создании объекта его тип совпадает с T.

В теле метода main() содержится выражение, иллюстрирующее невозможность объявить массив ссылок на уточненный универсальный тип. Строка, приведенная ниже, не будет скомпилирована.

// Gen<Integer> gens[] = new Gen<Integer>[10]; // Ошибка!

Универсальный класс не может расширять класс Throwable. Это означает, что создать универсальный класс исключения невозможно.

Лямбда выражения

Общая форма записи.

(параметры) -> (тело)

(Object arg1, Object arg2) -> arg1.equals(arg2);

Лямбда выражение содержит три части: список параметров, стрелка, тело.

1. Лямбда выражение может содержать ноль и более входных параметров.

```
(int a1, int a2) -> { return a1 - a2; }
(String s) -> { System.out.println(s); }
() -> 89;
```

2. Тип параметра может быть явно объявлен или выведен компилятором из значения параметра.

```
(String s) -> { System.out.println(s); }
```

Можно переписать в другую форму.

```
(s) -> { System.out.println(s); }
```

3. Если параметров нет или параметров больше одного, скобки необходимы.

```
(a1, a2) -> return a1 + a2;
(int a1, int a2) -> return a1 + a2;
() -> 42;
```

4. Нет необходимости объявлять один параметр в скобках, но в этом случае нельзя явно указать тип параметра.

```
a1 -> return 2 * a1
```

5. Тело лямбда выражения может содержать одно и более выражений.

6. Нет необходимости использовать фигурные скобки и ключевое слово return, если тело состоит из одного выражения.

```
() -> 4;
(int a) -> a * 6;
```

7. Если тело содержит более одного выражения, фигурные скобки и ключевое слово return необходимы.

```
() -> {
    System.out.println("Hi");
    return 4;
}
(int a) -> {
    System.out.println(a);
    return a * 6;
}
```

8. Если ключевое слово return отсутствует, возвращаемый тип может быть void.

```
() -> System.out.println("Hi");
() -> {
    System.out.println("Hi");
    return;
}
```

Функциональный интерфейс

Чтобы иметь возможность использовать лямбда выражения, необходим интерфейс.

```
public interface Searchable {
    boolean test(Car car);
}
```

```
Searchable s = (Car c) -> c.getCostUSD() > 20000;
```

Лямбда выражения не содержат информацию о том, какой функциональный интерфейс они реализуют.

Тип выражения выводится из контекста, в котором используется лямбда выражение. Этот тип называется целевой тип (target type).

Если лямбда выражение присваивается какому-то интерфейсу, лямбда выражение должно соответствовать синтаксису метода интерфейса.

Одно и то же лямбда выражение может использоваться с разными интерфейсами, если они имеют абстрактные методы, которые совместимы.

```
interface Searchable {
    boolean test(Car car);
}
interface Saleable {
    boolean approve(Car car);
}
    //...
    Searchable s1 = c -> c.getCostUSD() > 20000;
    Saleable s2 = c -> c.getCostUSD() > 20000;
```

Функциональный интерфейс (functional interface) – это интерфейс у которого только один абстрактный метод. Функциональный интерфейс может содержать любое количество методов по умолчанию (default) или статических методов.

```
interface A {
    default int defaultMethod() {
        return 0;
    }
    void method();
}
```

```
interface B {
    default int defaultMethod() {
        return 0;
    }
    default int anotherDefaultMethod() {
        return 0;
    }
    void method();
}
```

Функциональный интерфейс может содержать методы класса Object.

```
interface A {
    boolean equals(Object o);
    int hashCode();
    String toString();
    void method();
}
```

Встроенные функциональные интерфейсы

В Java 8 добавлены встроенные функциональные интерфейсы в пакет java.util.function:

- Predicate.
- Consumer.
- Function.
- Supplier.
- UnaryOperator.

Ссылки на методы

Если лямбда выражения вызывают только один существующий метод, лучше ссылать на этот метод по его имени. *Ссылки на методы* (Method References) – это компактные лямбда выражения для методов, у которых уже есть имя.

```
Consumer<String> consumer = str -> System.out.println(str);
// Можно переписать с помощью method references
Consumer<String> consumer = System.out::println;
```

Ссылки на методы бывают четырех видов.

Тип	Пример
Ссылка на статический метод	ContainingClass::staticMethodName
Ссылка на нестатический метод конкретного объекта	containingObject::instanceMethodName
Ссылка на нестатический метод любого объекта конкретного типа	ContainingType::methodName
Ссылка на конструктор	ClassName::new

Задание

Необходимо написать лямбда выражение, которое принимает на вход число и возвращает значение "Положительное число", "Отрицательное число" или "Ноль". Используем функциональный интерфейс Function.

Коллекции

Пакет java.util содержит одно из наиболее захватывающих расширений – *коллекции*. Коллекция – это группа объектов. Добавление коллекций вызвало фундаментальные изменения в структуре и архитектуре многих элементов пакета java.util. Они также расширили круг задач, к которым может применяться пакет.

Структура коллекций (collections framework) Java стандартизирует способ, с помощью которого программы обрабатывают группы объектов.

Collection framework состоит из 3-х частей:

- интерфейсы;
- классы;
- алгоритмы.

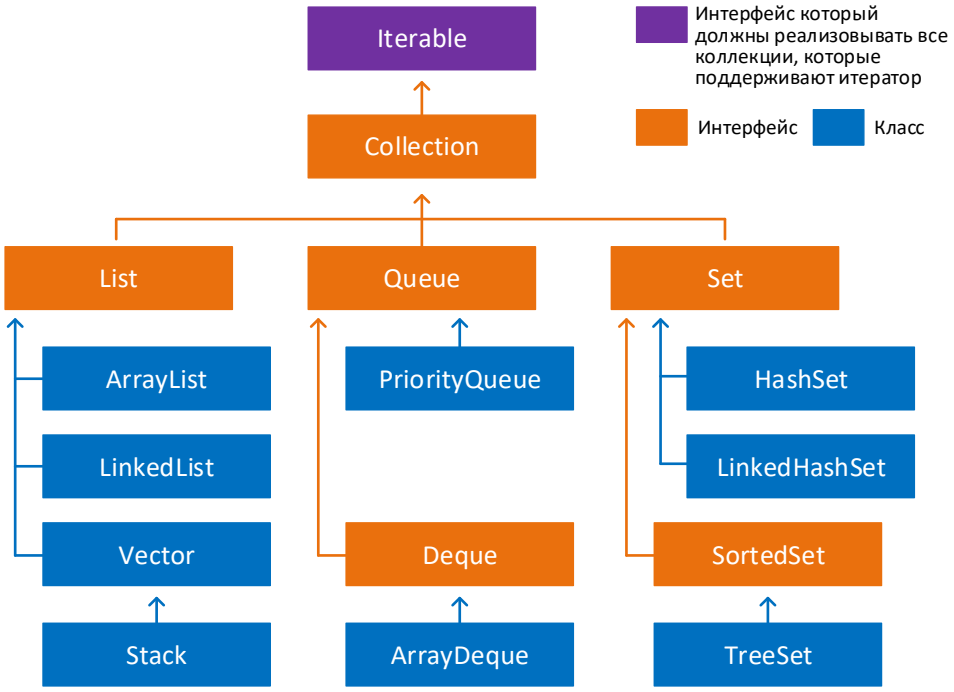
Интерфейсы представляют коллекции и позволяют манипулировать данными таким образом, чтобы создавать собственные коллекции для специфичных задач.

Классы, которые реализуют интерфейсы – это конкретные реализации интерфейсов, которые представляют собой структуры данных.

Алгоритмы – это наборы методов, которые позволяют эффективно выполнять такие операции, как поиск и сортировка в структурах данных Collections Framework.

Элемент, созданный структурой коллекций, – это интерфейс Iterator. Итератор обеспечивает универсальный, стандартизированный способ доступа к элементам коллекции – по одному. Таким образом, итератор обеспечивает средства перечисления содержимого коллекции.

Кроме итератора есть еще интерфейс Iterable – его должны реализовывать все коллекции, которые поддерживают итератор.



Интерфейс Collection

Интерфейс Collection – это основа, на которой сформирована структура коллекций. В нем объявляются основные методы, которые будут наследоваться всеми коллекциями.

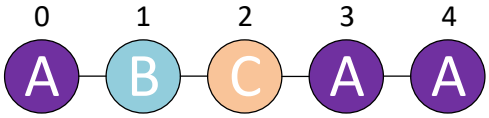
Методы интерфейса Collection:

- boolean add(E obj) – добавляет obj к вызывающей коллекции. Возвращает true, если obj был добавлен к коллекции.
- boolean addAll(Collection<? extends E> c) – добавляет все элементы к вызывающей коллекции. Возвращает true, если операция удалась (т.е. все элементы добавлены). В противном случае возвращает false.
- void clear() – удаляет все элементы вызывающей коллекции.
- boolean contains(Object obj) – возвращает true, если obj является элементом вызывающей коллекции. В противном случае возвращает false.
- boolean containsAll(Collection<?> c) – возвращает true, если вызывающая коллекция содержит все элементы c. В противном случае возвращает false.
- boolean equals(Object obj) – возвращает true, если вызывающая коллекция и obj эквивалентны. В противном случае возвращает false.
- int hashCode() – возвращает хешкод вызывающей коллекции.
- boolean isEmpty() – возвращает true, если вызывающая коллекция пуста. В противном случае возвращает false.
- Iterator<E> iterator() – возвращает итератор для вызывающей коллекции.
- boolean remove(Object obj) – удаляет один экземпляр obj из вызывающей коллекции. Возвращает true, если элемент удален. В противном случае возвращает false.
- boolean removeAll(Collection<?> c) – удаляет все элементы из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (т.е. элементы удалены). В противном случае возвращает false.
- boolean retainAll(Collection<?> c) – удаляет все элементы кроме входящих из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (т.е. элементы удалены). В противном случае возвращает false.
- int size() – возвращает количество элементов, содержащихся в коллекции.
- Object[] toArray() – возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции.
- removeIf(Predicate<? super E> filter) – удаляет элементы из коллекции, соответствующие заданному условию.

Структуры данных

Список – это упорядоченный набор элементов, для каждого из которых хранится указатель на следующий (для двусвязного списка и на следующий, и на предыдущий) элементы списка. Элементы в списке могут повторяться. Одна из важных особенностей списка – это наличие индекса у каждого элемента.

На рисунке ниже представлена коллекция, в которую добавлены буквы A, B, C, A, A. Буква A добавлена несколько раз – это разрешается в списках. Каждая буква отмечена индексом.



Стек – это коллекция, элементы которой получают по принципу "последний вошел, первый вышел" (Last-In-First-Out или LIFO). Это значит, что будет доступ

только к последнему добавленному элементу. Стек разрешает повтор одинаковых элементов.

Хорошим примером стека является стопка тарелок – верхняя тарелка была добавлена последняя в стопку (Last-In), но забрать ее можно будет первой (First-Out). Самая нижняя тарелка напротив была добавлена первой, но убрать ее можно будет только после удаления всех верхних тарелок. Получить доступ к тарелкам в середине стопки нельзя без удаления верхних тарелок.

Очереди очень похожи на стеки. Они также не дают доступа к произвольному элементу, но, в отличие от стека, элементы помещаются (enqueue) и забираются (dequeue) с разных концов. У очереди выделяют хвост и голову. Такой метод называется "первый вошел, первый вышел" (First-In-First-Out или FIFO). Забирать элементы из очереди можно будет в том же порядке, что и при помещении их в очередь. Очереди также разрешают повторы элементов.

Множество – неупорядоченный набор элементов, без повторов. Хорошим примером множества являются автомобили на парковке – каждое авто имеет уникальный номер, и нет необходимости их упорядочивать.

Интерфейс List

Интерфейс List расширяет Collection и объявляет поведение коллекции, которая хранит последовательность элементов. Элементы могут быть вставлены или извлечены с помощью их позиций в списке через отсчитываемый от нуля индекс. Список может содержать дублированные элементы. В дополнение к методам, определенным в Collection, интерфейс List определяет собственные методы:

- void add(int index, E obj) – вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. Т.е. никакие элементы не перезаписываются.
- boolean addAll (int index,Collection<? extends E> c) – вставляет все элементы в вызывающий список, начиная с позиции, переданной в index. Все ранее существовавшие элементы за точкой вставки смещаются вверх. Т.е. никакие элементы не перезаписываются. Возвращает true, если вызывающий список изменяется, и false в противном случае.
- E get (int index) – возвращает объект, сохраненный в указанной позиции вызывающего списка.
- int indexOf(Object obj) – возвращает индекс первого экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается 1.
- int lastIndexOf(Object obj) – возвращает индекс последнего экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается 1.
- ListIterator listIterator() – возвращает итератор, указывающий на начало списка.
- ListIterator listIterator(int index) – возвращает итератор, указывающий на заданную позицию в списке.
- E remove(int index) – удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад.
- E set (int index, E obj) – присваивает obj элементу, находящемуся в списке в позиции index.
- default void sort(Comparator<? super E> c) – сортирует список, используя заданный компаратор.
- List subList (int start, int end) – возвращает список, включающий элементы от start до end–1 из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке.

Класс ArrayList

Одной из реализаций интерфейса List является класс ArrayList. Он поддерживает динамические массивы, которые могут расти по мере необходимости. Элементы ArrayList могут быть абсолютно любых типов, в том числе и null. Элементы в этом виде коллекции могут повторяться. Данный класс используется чаще всех остальных реализаций коллекции.

Объект класса ArrayList, содержит свойства elementData и size. Хранилище значений elementData есть не что иное, как массив определенного типа. По умолчанию размер этого внутреннего массива будет иметь 16 элементов (при создании класса конструктором без параметров). Если добавить в ArrayList больше 16 элементов, ничего плохого не произойдет (в отличие от массивов, где будет выброшено ArrayIndexOutOfBoundsException исключение). В этом случае просто произойдет пересоздание внутреннего массива elementData, и это произойдет неявно.

В случае если заранее известно максимальное количество элементов в создаваемой коллекции, размер массива можно указать, передав нужное значение в конструктор ArrayList. Такая программа не будет тратить ресурсы на пересоздание внутреннего массива и будет работать немного быстрее.

У класса ArrayList есть следующие конструкторы:

- ArrayList() – помогает создать пустую коллекцию с внутренним массивом размер которого будет 16 элементов.
- ArrayList(Collection <? extends E> collection) – создает коллекцию и заполняет ее элементами из передаваемой коллекции collection.
- ArrayList(int capacity) – помогает создать пустую коллекцию с внутренним массивом, размер которого будет равен значению параметра capacity.

Достоинства класса ArrayList:

- Быстрый доступ по индексу. Скорость операции.
- Быстрая вставка и удаление элементов с конца. Скорость операций.

Недостатки класса ArrayList:

- Медленная вставка и удаление элементов из середины. Такие операции имеют большую сложность. Если придется выполнять достаточно много операций такого типа, лучше выбрать другой класс.

Для удобства в следующем списке приведены все методы, позволяющие добавлять элементы в объект класса ArrayList:

- boolean add(E obj) – добавляет obj к вызывающей коллекции. Возвращает true, если obj был добавлен к коллекции (интерфейс Collection).
- void add(int index, E obj) – вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. Т.е. никакие элементы не перезаписываются (интерфейс List).
- E set (int index, E obj) – присваивает obj элементу, находящемуся в списке в позиции index (интерфейс List).
- boolean addAll (Collection<? extends E> c) – добавляет все элементы к вызывающей коллекции. Возвращает true, если операция удалась (т.е. все элементы добавлены). В противном случае возвращает false (интерфейс Collection).

В следующем примере рассмотрим создание объекта класса ArrayList и добавление в него элементов с помощью указанных выше методов

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListAddDemo {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        System.out.println("Начальный размер arrayList: " + arrayList.size());
        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        arrayList.add("F");
        arrayList.add( index: 1,  element: "A2");
        arrayList.set(0, "C2");
        System.out.println("Размер arrayList после добавления: " + arrayList.size());
        System.out.println("Содержимое arrayList: " + arrayList);
        System.out.println(arrayList.get(0));
    }
}
```

```
Начальный размер arrayList: 0
Размер arrayList после добавления: 8
Содержимое arrayList: [C2, A2, A, E, B, D, F, F]
C2
```

Методы класса ArrayList для удаления элементов:

- boolean remove(Object obj) – удаляет один экземпляр obj из вызывающей коллекции. Возвращает true, если элемент удален. В противном случае возвращает false (интерфейс Collection).
- E remove(int index) – удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, т.е. элементы, следующие за удаленным, сдвигаются на одну позицию назад (интерфейс List).
- boolean removeAll(Collection<?> c) – удаляет все элементы из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (т.е. элементы удалены). В противном случае возвращает false (интерфейс Collection).

- boolean retainAll(Collection<?> c) – удаляет все элементы кроме входящих из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (т.е. элементы удалены). В противном случае возвращает false (интерфейс Collection).
- void clear() – удаляет все элементы вызывающей коллекции (интерфейс Collection).

Ниже приведен пример удаления элементов из ArrayList.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListRemoveDemo {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        arrayList.add("F");
        arrayList.add( index: 1,  element: "A2");
        arrayList.set(0, "C2");
        System.out.println("Содержимое arrayList: " + arrayList);
        System.out.println("Размер arrayList после добавления: " + arrayList.size());
        arrayList.remove( o: "F");
        arrayList.remove( index: 2);
        System.out.println("Размер arrayList после удаления: " + arrayList.size());
        System.out.println("Содержимое of arrayList: " + arrayList);
    }
}
```

```
Содержимое arrayList: [C2, A2, A, E, B, D, F, F]
Размер arrayList после добавления: 8
Размер arrayList после удаления: 6
Содержимое of arrayList: [C2, A2, E, B, D, F]
```

В следующем коде приведен пример использования метода removeAll() класса ArrayList.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListRemoveAllDemo {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        arrayList.add("F");
        arrayList.add( index: 1,  element: "A2");
        arrayList.set(0, "C2");
        List<String> removeElements = List.of( e1: "C2",  e2: "A2",  e3: "AA",  e4: "F");
        System.out.println("Содержимое arrayList до removeAll: " + arrayList);
        arrayList.removeAll( c: removeElements);
        System.out.println("Содержимое arrayList после removeAll: " + arrayList);
    }
}
```

```
Содержимое arrayList до removeAll: [C2, A2, A, E, B, D, F, F]
Содержимое arrayList после removeAll: [A, E, B, D]
```

В следующем коде приведен пример использования методов addAll(), clear() класса ArrayList.

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<String> arrayList1 = new ArrayList<>();
        List<String> arrayList2 = List.of( e1: "1",  e2: "2");
        arrayList1.add("A");
        arrayList1.add("B");
        arrayList1.add("C");
        arrayList1.add("D");
        arrayList1.add("E");
        arrayList1.add("F");
        System.out.println("arrayList1 до добавления " + arrayList1);
        arrayList1.addAll( index: 3,  c: arrayList2);
        System.out.println("arrayList1 после добавления " + arrayList1);
        arrayList1.clear();
        System.out.println("arrayList1 после очистки " + arrayList1);
    }
}
```

```
arrayList1 до добавления [A, B, C, D, E, F]
arrayList1 после добавления [A, B, C, 1, 2, D, E, F]
arrayList1 после очистки []
```

В следующем коде приведен пример использования метода retainAll() класса ArrayList.

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<String> arrayList1 = new ArrayList<>();
        List<String> arrayList2 = List.of( e1: "F",  e2: "FF",  e3: "E");
        arrayList1.add("A");
        arrayList1.add("A");
        arrayList1.add("B");
        arrayList1.add("C");
        arrayList1.add("D");
        arrayList1.add("E");
        arrayList1.add("F");
        arrayList1.add("F");
        arrayList1.retainAll( c: arrayList2);
        System.out.println(arrayList1);
    }
}
```

```
[E, F, F]
```

Достаточно частая задача – это получение массива из коллекции. Для этого в интерфейсе Collection объявлен метод toArray(). Есть два варианта метода toArray().

Первый это Object[] toArray(), он создает массив типа Object и записывает в него значения из вызывающей коллекции.

```java
import java.util.Arrays;
import java.util.List;

public class ArrayListToStringDemo {
    public static void main(String[] args) {
        List<String> arrayList = List.of( e1: "C",  e2: "A",  e3: "E",  e4: "B",  e5: "D",  e6: "F");
        Object[] objectArray = arrayList.toArray();
        System.out.println(Arrays.toString( a: objectArray));
    }
}
```

```
[C, A, E, B, D, F]
```

Такой вариант не очень удобен тем, что массив имеет тип Object. Поэтому чаще используется следующий перегруженный вариант.

<T>T[] toArray(T массив[]) – создает массив такого же типа, как и вызывающая коллекция. Этот метод имеет два варианта использования. Первый вариант: можно создать массив такого же типа, как и вызывающая коллекция и передать этот массив в метод toArray(). И второй вариант: в метод передается

массив того же типа, что и вызывающей коллекции, но размером 0. Метод создает новый массив и записывает в него значения из коллекции.

```
[C, A, E, B, D, F]
[C, A, E, B, D, F]
```

```java
import java.util.Arrays;
import java.util.List;

public class ArrayListToStringDemo {
    public static void main(String[] args) {
        List<String> arrayList = List.of( e1: "C",  e2: "A",  e3: "E",  e4: "B",  e5: "D",  e6: "F");
        // 1 вариант
        String[] stringArray1 = new String[arrayList.size()];
        arrayList.toArray( a: stringArray1);
        System.out.println(Arrays.toString( a: stringArray1));

        // 2 вариант
        String[] stringArray2 = arrayList.toArray( a: new String[0]);
        System.out.println(Arrays.toString( a: stringArray2));
    }
}
```

Интерфейс Set

Интерфейс Set определяет множество (набор). Он расширяет Collection и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод add() возвращает false, если делается попытка добавить дублированный элемент в набор.

Интерфейс не определяет никаких собственных дополнительных методов.

Интерфейс Set заботится об уникальности хранимых объектов, уникальность определятся реализацией метода equals(). Поэтому если объекты создаваемого класса будут добавляться в Set, желательно переопределить метод equals().

Класс HashSet

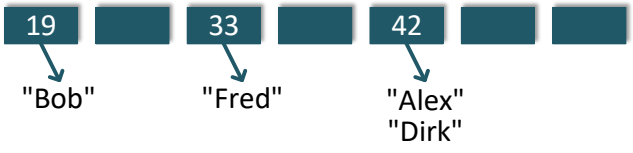
Класс HashSet реализует интерфейс Set и создает коллекцию, которая хранит элементы в хеш-таблице.

Элементы хеш-таблицы хранятся в виде пар ключ-значение. Ключ определяет ячейку (или бакет) для хранения значения. Содержимое ключа служит для определения однозначного значения, называемого хеш-кодом.

Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом.

Рассмотрим пример вычисления хеш-кодов.

Ключ	Алгоритм хеш-кода	Хеш-код
Alex	A(1) + l(12) + e(5) + x(24)	42
Bob	B(2) + o(15) + b(2)	19
Dirk	D(4) + i(9) + r(18) + k(11)	42
Fred	F(6) + r(18) + e(5) + d(4)	33



Допустим, чтобы получить хеш-код, нужно сопоставить каждый символ из ключа с номером этого символа в алфавите и получить их сумму. Таким образом "Bob" попадает в ячейку 19, "Fred" в ячейку 33. Как видно из примера, для ключей "Alex" и "Dirk" получается одинаковый хеш-код, что тоже допустимо.

Такая ситуация называется коллизией, два значения попадают в одну ячейку с номером 42.

Обратный процесс – поиск элемента в хеш-таблице происходит следующим образом. Есть ключ объекта, по которому вычисляется хеш-код и определяется ячейка, в котором находится объект. Если в ячейке находится не один, а несколько элементов необходимо использовать метод equals() для определения нужного объекта.

Методы hashCode() и equals() тесно связаны между собой. Поэтому существуют специальный контракт написания методов hashCode() и equals():

- Для одного и того же объекта, хеш-код всегда будет одинаковым.
- Если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот).
- Если хеш-коды равны, то входные объекты не всегда равны.
- Если хеш-коды разные, то и объекты гарантированно будут разные.

На самом деле, корректным, но не эффективным может быть такое определение хеш-кода.

```
public int hashCode() {
   return 1;
}
```

Это определение метода hashCode() соответствует контракту, но он разместит все объекты класса в одной ячейке, что нивелирует все достоинства хеш-таблицы.

Создание метода hashCode()

Эффективным алгоритмом для определения хеш-кода объекта является такой алгоритм, который ровным слоем распределяет объекты по хеш-таблице.

В средствах разработки, в частности в IntelliJ IDEA, существует генератор, который сгенерирует метод hashCode(). Обычно все пользуются этим вариантом создания метода.

Конструкторы класса HashSet:

- HashSet() – начальная емкость по умолчанию (initialCapacity) – 16, коэффициент загрузки по умолчанию (loadFactor) – 0,75.
- HashSet(int initialCapacity) – коэффициент загрузки – 0,75.
- HashSet(int initialCapacity, float loadFactor)
- HashSet(Collection collection) – конструктор, добавляющий элементы из другой коллекции.

В конструкторах можно указывать такие параметры как начальная емкость и коэффициент загрузки.

Начальная емкость (initial capacity) – это изначальное количество ячеек (buckets) в хэш-таблице. Если все ячейки будут заполнены, их количество увеличится автоматически.

Коэффициент загрузки (load factor) – это показатель того, насколько заполненным может быть HashSet до того момента, когда его емкость автоматически увеличится. Когда количество элементов в HashSet становится больше, чем capacity * loadfactor, хэш-таблица ре-хэшируется (заново вычисляются хэш-коды элементов, и таблица перестраивается согласно полученным значениям) и количество ячеек в ней увеличивается в два раза.

Коэффициент загрузки, равный 0,75, в среднем обеспечивает хорошую производительность. Если этот параметр увеличить, тогда уменьшится нагрузка на память (так как это уменьшит количество операций ре-хэширования и перестраивания), но это повлияет на операции добавления и поиска. Чтобы

минимизировать время, затрачиваемое на ре-хэширование, нужно правильно подобрать параметр начальной емкости.

Достоинства и недостатки класса HashSet:

- Выгода от хеширования состоит в том, что оно обеспечивает постоянство время выполнения операций add(), contains(), remove() и size(), даже для больших наборов.
- Недостаток класса HashSet (или можно даже сказать особенность) в том, что он не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств.

В следующем коде приведен пример использования класса HashSet.

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();

        set.add("Минск");
        set.add("Борисов");
        set.add("Витебск");
        set.add("Гродно");
        set.add("Минск");
        System.out.println(set);
    }
}
```

```
[Гродно, Борисов, Минск, Витебск]
```

Класс LinkedHashSet

Класс LinkedHashSet расширяет HashSet, не добавляя никаких новых методов.

LinkedHashSet поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. Но это приводит к тому что класс LinkedHashSet выполняет операции дольше чем класс HashSet.

В следующем коде приведен пример использования класса LinkedHashSet.

```
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetDemo {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<>();
        set.add("Бета");
        set.add("Альфа");
        set.add("Эта");
        set.add("Гамма");
        set.add("Эпсилон");
        set.add("Омега");
        System.out.println(set);
    }
}
```

```
[Бета, Альфа, Эта, Гамма, Эпсилон, Омега]
```

Интерфейс SortedSet

Интерфейс SortedSet, расширяющий интерфейс Set, описывает упорядоченное множество, отсортированное в возрастающем порядке или по порядку, заданному реализацией интерфейса Comparator.

Методы интерфейса SortedSet:

- Comparator<? super E> comparator() – возвращает компаратор сортированного множества. Если для множества применяется естественный порядок сортировки, возвращается null.

- E first() – возвращает первый элемент вызывающего сортированного множества.
- E last() – возвращает последний элемент вызывающего сортированного множества.
- SortedSet headSet(E toElement) – возвращает SortedSet, содержащий элементы из вызывающего множества, которые предшествуют end.
- SortedSet subSet(E fromElement, E toElement) – возвращает SortedSet, содержащий элементы из вызывающего множества, находящиеся между start и end–1.
- SortedSet tailSet(E fromElement) – возвращает SortedSet, содержащий элементы из вызывающего множества, которые следуют за end.

В следующем коде приведен пример использования методов subSet(), headSet(), tailSet(), first(), last().

```
import java.util.SortedSet;
import java.util.TreeSet;

public class TreeSetDemo {
    public static void main(String[] args) {
        SortedSet<String> set = new TreeSet<>();
        set.add("Минск");
        set.add("Борисов");
        set.add("Витебск");
        set.add("Брест");
        set.add("Минск");
        System.out.println(set);
        SortedSet<String> subSet = set.subSet("Борисов", "Витебск");
        System.out.println("SubSet: " + subSet);
        System.out.println("HeadSet: " + set.headSet( toElement: "Витебск"));
        System.out.println("TailSet: " + set.tailSet( fromElement: "Витебск"));
        System.out.println("Первый элемент: " + set.first());
        System.out.println("Последний элемент: " + set.last());
    }
}
```

```
[Борисов, Брест, Витебск, Минск]
SubSet: [Борисов, Брест]
HeadSet: [Борисов, Брест]
TailSet: [Витебск, Минск]
Первый элемент: Борисов
Последний элемент: Минск
```

Класс TreeSet

Класс TreeSet реализует интерфейс NavigableSet, который поддерживает элементы в отсортированном по возрастанию порядке. Объекты сохраняются в отсортированном порядке по возрастанию.

Обработка операций удаления и вставки объектов происходит медленнее чем в хэш-множествах, но быстрее, чем в списках.

Конструкторы класса TreeSet:

- TreeSet().
- TreeSet(Collection<? extends E> collection).
- TreeSet(Comparator<? super E> comparator).
- TreeSet(SortedSet<E> sortedSet).

Интерфейсы Comparable и Comparator

Существует два способа сравнения объектов:

- С помощью интерфейса Comparable.
- С помощью интерфейса Comparator.

Интерфейс Comparable

Для того чтобы объекты можно было сравнить и сортировать, они должны реализовать параметризованный интерфейс Comparable.

Интерфейс Comparable содержит один единственный метод int compareTo(T item), который сравнивает текущий объект с объектом, переданным в качестве параметра.

Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвращает ноль, значит, оба объекта равны.

```java
public class Person implements Comparable<Person> {
    private String firstName;
    private String lastName;
    private int age;
    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int compareTo(Person anotherPerson) {
        int anotherPersonAge = anotherPerson.getAge();
        return this.age - anotherPersonAge;
    }
    public String toString() {
        return "Person{" +
                "firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", age=" + age +
                '}';
    }
```

```java
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        if (getAge() != person.getAge()) return false;
        if (getFirstName() != null ? !getFirstName().equals(person.getFirstName()) : person.getFirstName() != null)
            return false;
        return getLastName() != null ? getLastName().equals(person.getLastName()) : person.getLastName() == null;
    }
    public int hashCode() {
        int result = getFirstName() != null ? getFirstName().hashCode() : 0;
        result = 31 * result + (getLastName() != null ? getLastName().hashCode() : 0);
        result = 31 * result + getAge();
        return result;
    }
}
```

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class ComparePersonDemo {
    public static void main(String[] args) {
        SortedSet<Person> set = new TreeSet<>();
        set.add(new Person( firstName: "Саша",  lastName: "Иванов",  age: 36));
        set.add(new Person( firstName: "Маша",  lastName: "Петрова",  age: 23));
        set.add(new Person( firstName: "Даша",  lastName: "Сидорова",  age: 34));
        set.add(new Person( firstName: "Вася",  lastName: "Иванов",  age: 25));
        set.forEach( action: System.out::println);
    }
}
```

```
Person{firstName='Маша', lastName='Петрова', age=23}
Person{firstName='Вася', lastName='Иванов', age=25}
Person{firstName='Даша', lastName='Сидорова', age=34}
Person{firstName='Саша', lastName='Иванов', age=36}
```

Интерфейс Comparator

Если класс по какой-то причине не может реализовать интерфейс Comparable, или же просто нужен другой вариант сравнения, используется интерфейс Comparator.

Интерфейс содержит метод int compare(T o1, T o2), который должен быть реализован классом, реализующим компаратор.

Метод compare возвращает числовое значение – если оно отрицательное, то объект o1 предшествует объекту o2, иначе – наоборот. А если метод возвращает ноль, то объекты равны.

Для применения интерфейса вначале надо создать класс компаратора, который реализует этот параметризованный интерфейс.

```
import java.util.Comparator;

public class PersonComparator implements Comparator<Person> {
    public int compare(Person o1, Person o2) {
        return o1.getLastName().compareTo(o2.getLastName());
    }
}
```

```
import java.util.SortedSet;
import java.util.TreeSet;

public class PersonComparatorDemo {
    public static void main(String[] args) {
        PersonComparator personComparator = new PersonComparator();
        SortedSet<Person> set = new TreeSet<>(personComparator);
        set.add(new Person( firstName: "Саша",  lastName: "Иванов",  age: 36));
        set.add(new Person( firstName: "Маша",  lastName: "Петрова",  age: 23));
        set.add(new Person( firstName: "Даша",  lastName: "Сидорова",  age: 34));
        set.add(new Person( firstName: "Вася",  lastName: "Иванов",  age: 25));
        // Было добавлено 4 элемента, но распечатано 3
        set.forEach( action: System.out::println);
    }
}
```

```
Person{firstName='Саша', lastName='Иванов', age=36}
Person{firstName='Маша', lastName='Петрова', age=23}
Person{firstName='Даша', lastName='Сидорова', age=34}
```

Перепишем код с использованием метода comparing() интерфейса Comparator.

```
import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;

public class PersonComparatorDemo {
    public static void main(String[] args) {
        Comparator<Person> personComparator =
                Comparator.comparing( keyExtractor: Person::getLastName).thenComparing( keyExtractor: Person::getAge);
        SortedSet<Person> set = new TreeSet<>(personComparator);
        set.add(new Person( firstName: "Саша",  lastName: "Иванов",  age: 36));
        set.add(new Person( firstName: "Маша",  lastName: "Петрова",  age: 23));
        set.add(new Person( firstName: "Даша",  lastName: "Сидорова",  age: 34));
        set.add(new Person( firstName: "Вася",  lastName: "Иванов",  age: 25));
        set.forEach( action: System.out::println);
    }
}
```

```
Person{firstName='Вася', lastName='Иванов', age=25}
Person{firstName='Саша', lastName='Иванов', age=36}
Person{firstName='Маша', lastName='Петрова', age=23}
Person{firstName='Даша', lastName='Сидорова', age=34}
```

Интерфейс NavigableSet

Интерфейс NavigableSet расширяет SortedSet и добавляет методы для более удобного поиска по коллекции:

- E ceiling(E obj) – ищет в наборе наименьший элемент e, для которого истинно e >= obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.

- E floor(E obj) – ищет в наборе наибольший элемент e, для которого истинно e <= obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
- E higher(E obj) – ищет в наборе наибольший элемент e, для которого истинно e > obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
- E lower(E obj) – ищет в наборе наименьший элемент e, для которого истинно e < obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
- NavigableSet headSet(E upperBound, boolean incl) – возвращает NavigableSet, включающий все элементы вызывающего набора, меньшие upperBound. Результирующий набор поддерживается вызывающим набором (backed-collection).
- NavigableSet subSet(E lowerBound, boolean lowIncl, E upperBound, boolean highIncl) – возвращает NavigableSet, включающий все элементы вызывающего набора, которые больше lowerBound и меньше upperBound. Если lowIncl равно true, то элемент, равный lowerBound, включается. Если highIncl равно true, также включается элемент, равный upperBound.
- NavigableSet tailSet(E fromElement, boolean inclusive) – возвращает NavigableSet, включающий все элементы вызывающего набора, которые больше (или равны, если inclusive равно true) чем fromElement. Результирующий набор поддерживается вызывающим набором (backed-collection).
- E pollLast() – возвращает последний элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наибольшим значением. Возвращает null в случае пустого набора.
- E pollFirst() – возвращает первый элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наименьшим значением. Возвращает null в случае пустого набора.
- Iterator descendingIterator() – возвращает итератор, перемещающийся от большего к меньшему, другими словами, обратный итератор.
- NavigableSet descendingSet() – возвращает NavigableSet, представляющий собой обратную версию вызывающего набора. Результирующий набор поддерживается вызывающим набором.

Класс, реализующие интерфейс NavigableSet – это TreeSet.

В следующем коде приведен пример использования интерфейса NavigableSet.

```
import java.util.NavigableSet;
import java.util.SortedSet;
import java.util.TreeSet;

public class Ferry {
    public static void main(String[] args) {
        NavigableSet<Integer> times = new TreeSet<>();
        times.add(1205);
        times.add(1505);
        times.add(1545);
        times.add(1830);
        times.add(2010);
        times.add(2100);
        // Java 5 версия
        System.out.println("Java 5");
        SortedSet<Integer> subset = times.headSet( toElement: 1600);
        System.out.println("Последний паром перед 16:00 - " + subset.last());

        SortedSet<Integer> tailSet = times.tailSet( fromElement: 2000);
        System.out.println("Первый паром после 20:00 - " + tailSet.first());
        System.out.println();

        // Java 6 версия использует новые методы lower() и higher()
        System.out.println("Java 6");
        System.out.println("Последний паром перед 16:00 - " + times.lower( e: 1600));
        System.out.println("Первый паром после 20:00 - " + times.higher( e: 2000));
    }
}
```

```
Java 5
Последний паром перед 16:00 - 1545
Первый паром после 20:00 - 2010

Java 6
Последний паром перед 16:00 - 1545
Первый паром после 20:00 - 2010
```

Интерфейс Queue и классы

Интерфейс Queue расширяет Collection и объявляет поведение очередей, которые представляют собой список "первый вошел, первый вышел" (FIFO). Существуют разные типы очередей, в которых порядок основан на некотором критерии. Очереди не могут хранить значения null.

Методы интерфейса Queue:

- E element() – возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, инициируется исключение NoSuchElementException.
- E remove() – удаляет элемент из головы очереди, возвращая его. Инициирует исключение NoSuchElementException, если очередь пуста.
- E peek() – возвращает элемент из головы очереди. Возвращает null, если очередь пуста. Элемент не удаляется.
- E poll() – возвращает элемент из головы очереди и удаляет его. Возвращает null, если очередь пуста.
- boolean offer(E obj) – пытается добавить obj в очередь. Возвращает true, если obj добавлен, и false в противном случае.

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.offer(e: "Минск");
        queue.offer(e: "Бобруйск");
        queue.offer(e: "Брест");
        queue.offer(e: "Могилев");
        System.out.println(queue.peek());
        String town;
        while ((town = queue.poll()) != null) {
            System.out.println(town);
        }
    }
}
```

```
Минск
Минск
Бобруйск
Брест
Могилев
```

Интерфейс Deque

Интерфейс Deque расширяет Queue и описывает поведение двунаправленной очереди. Двунаправленная очередь может функционировать как стандартная очередь FIFO либо как стек LIFO.

Методы интерфейса Deque:

- void addFirst(E obj) – добавляет obj в голову двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.
- void addLast(E obj) – добавляет obj в хвост двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.
- E getFirst() – возвращает первый элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключение NoSuchElementException.
- E getLast() – возвращает последний элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключения NoSuchElementException.
- boolean offerFirst(E obj) – пытается добавить obj в голову двунаправленной очереди. Возвращает true, если obj добавлен, и false в противном случае. Таким образом, этот метод возвращает false при попытке добавить obj в полную двунаправленную очередь ограниченной емкости.
- boolean offerLast(E obj) – пытается добавить obj в хвост двунаправленной очереди. Возвращает true, если obj добавлен, и false в противном случае.

- E pop() – возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.
- void push(E obj) – добавляет элемент в голову двунаправленной очереди. Если в очереди фиксированного объема нет места, возбуждает исключение IllegalStateException.
- E peekFirst() – возвращает элемент, находящийся в голове двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.
- E peekLast() – возвращает элемент, находящийся в хвосте двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.
- E pollFirst() – возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.
- E pollLast() – возвращает элемент, находящийся в хвосте двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.
- E removeLast() – возвращает элемент, находящийся в конце двунаправленной очереди, удаляя его в процессе. Возбуждает исключение NoSuchElementException, если очередь пуста.
- E removeFirst() – возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.
- boolean removeLastOccurrence(Object obj) – удаляет последнее вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false если очередь не содержала obj.
- boolean removeFirstOccurrence(Object obj) – удаляет первое вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false, если очередь не содержала obj.

Класс ArrayDeque

ArrayDeque создает двунаправленную очередь.

Конструкторы класса ArrayDeque:

- ArrayDeque() – создает пустую двунаправленную очередь с вместимостью 16 элементов.
- ArrayDeque(Collection<? extends E> c) – создает двунаправленную очередь из элементов коллекции c в том порядке, в котором они возвращаются итератором коллекции c.
- ArrayDeque(int numElements) – создает пустую двунаправленную очередь с вместимостью numElements.

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class ArrayDequeExample {
    public static void main(String[] args) {
        Deque<String> stack = new ArrayDeque<>();
        Deque<String> queue = new ArrayDeque<>( numElements: 2);
        stack.push( e: "A");
        stack.push( e: "B");
        stack.push( e: "C");
        stack.push( e: "D");
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
        System.out.println();
        queue.add("A");
        queue.add("B");
        queue.add("C");
        queue.add("D");
        while (!queue.isEmpty()) {
            System.out.print(queue.remove() + " ");
        }
    }
}
```

```
D C B A
A B C D
```

Класс LinkedList

Класс LinkedList реализует сразу два интерфейса – List, Deque. LinkedList – это двусвязный список.

Конструкторы класса LinkedList:

- LinkedList().
- LinkedList(Collection<? extends E> c).

Внутри класса LinkedList существует static inner класс Entry, с помощью которого создаются новые элементы

```
private static class Entry<E>
{
    E element;
    Entry<E> next;
    Entry<E> prev;
    Entry(E element, Entry<E> next, Entry<E> prev) {
        this.element = element;
        this.next = next;
        this.prev = prev;
    }
}
```

Из LinkedList можно организовать стек, очередь, или двойную очередь. На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время.

```
import java.util.LinkedList;

public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("F");
        list.add("B");
        list.add("D");
        list.add("E");
        list.add("C");
        list.addLast( e: "Z");
        list.addFirst( e: "A");
        list.add( index: 1, element: "A2");
        System.out.println("Содержимое: " + list);
        list.remove( o: "F");
        list.remove( index: 2);
        list.removeFirst();
        list.removeLast();
        System.out.println("Содержимое после удаления: " + list);
        String val = list.get(2);
        list.set(2, val + "+");
        System.out.println("Содержимое после изменения: " + list);
    }
}
```

```
Содержимое: [A, A2, F, B, D, E, C, Z]
Содержимое после удаления: [A2, D, E, C]
Содержимое после изменения: [A2, D, E+, C]
```

Класс PriorityQueue

PriorityQueue – это класс очереди с приоритетами. По умолчанию очередь с приоритетами размещает элементы согласно естественному порядку сортировки используя Comparable. Элементу с наименьшим значением присваивается наибольший приоритет. Если несколько элементов имеют одинаковый наивысший элемент – связь определяется произвольно. Также можно указать специальный порядок размещения, используя Comparator.

Конструкторы класса PriorityQueue:

- PriorityQueue() – создает очередь с приоритетами начальной емкостью 11, размещающую элементы согласно естественному порядку сортировки (Comparable).
- PriorityQueue(Collection<? extends E> c).
- PriorityQueue(int initialCapacity).

- PriorityQueue(int initialCapacity, Comparator<? super E> comparator).
- PriorityQueue(PriorityQueue<? extends E> c).
- PriorityQueue(SortedSet<? extends E> c).

```
Priority queue с Comparable: Брест Брест Витебск Гродно Минск
Priority queue с Comparator: Минск Гродно Витебск Брест Брест
```

```java
import java.util.Collections;
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        Queue<String> queue1 = new PriorityQueue<>();
        queue1.offer( e: "Минск");
        queue1.offer( e: "Гродно");
        queue1.offer( e: "Витебск");
        queue1.offer( e: "Брест");
        queue1.offer( e: "Брест");
        System.out.print("Priority queue с Comparable: ");
        while (queue1.size() > 0) {
            System.out.print(queue1.remove() + " ");
        }
        System.out.println();

        PriorityQueue<String> queue2 = new PriorityQueue<>( initialCapacity: 5,  comparator: Collections.reverseOrder());
        queue2.offer( e: "Минск");
        queue2.offer( e: "Гродно");
        queue2.offer( e: "Витебск");
        queue2.offer( e: "Брест");
        queue2.offer( e: "Брест");
        System.out.print("Priority queue с Comparator: ");
        while (queue2.size() > 0) {
            System.out.print(queue2.remove() + " ");
        }
    }
}
```

Интерфейс Iterator

Перебор содержимого коллекции может быть осуществлен двумя способами: с помощью цикла for each и с помощью итератора.

Итератор позволяет осуществлять обход коллекции и при желании удалять избранные элементы, используется интерфейс Iterator.

Чтобы получить объект итератора, необходимо вызвать метод Iterator<E> iterator().

Методы интерфейса Iterator:

- boolean hasNext() – возвращает true, если есть еще элементы. В противном случае возвращает false.
- E next() – возвращает следующий элемент. Если следующий элемент коллекции отсутствует, то метод next() генерирует исключение NoSuchElementException.
- void remove() – удаляет текущий элемент. Возбуждает исключение IllegalStateException, если предпринимается попытка вызвать remove(), которой не предшествовал вызов next().

```
C A E B D F
```

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IteratorDemo {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();
        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        Iterator<String> iterator = arrayList.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.print(element + " ");
        }
    }
}
```

Все классы в каркасе коллекций усовершенствованы таким образом, чтобы реализовывать интерфейс Iterable. Это означает, что содержимое коллекции можно перебрать, организовав цикл for в стиле for each. Конструкция for each скрывает итератор, поэтому нельзя вызвать метод remove().

Интерфейс ListIterator

Интерфейс ListIterator расширяет интерфейс Iterator и используется для двустороннего обхода списка и видоизменения его элементов.

ListIterator можно получить вызывая метод listIterator() для коллекций, реализующих List.

Методы интерфейса ListIterator:

- void add(E obj) – вставляет obj перед элементом, который должен быть возвращен следующим вызовом next().
- boolean hasNext() – возвращает true, если есть следующий элемент. В противном случае возвращает false.
- boolean hasPrevious() – возвращает true, если есть предыдущий элемент. В противном случае возвращает false.
- E next() – возвращает следующий элемент. Если следующего нет, инициируется исключение NoSuchElementException.
- int nextIndex() – возвращает индекс следующего элемента. Если следующего нет, возвращается размер списка.
- E previous() – возвращает предыдущий элемент. Если предыдущего нет, инициируется исключение NoSuchElementException.
- int previousIndex() – возвращает индекс предыдущего элемента. Если предыдущего нет, возвращается –1.
- void remove() – удаляет текущий элемент из списка. Если remove() вызван до next() или previous(), инициируется исключение IllegalStateException.
- void set(E obj) – присваивает obj текущему элементу. Это элемент, возвращенный последним вызовом next() или previous().

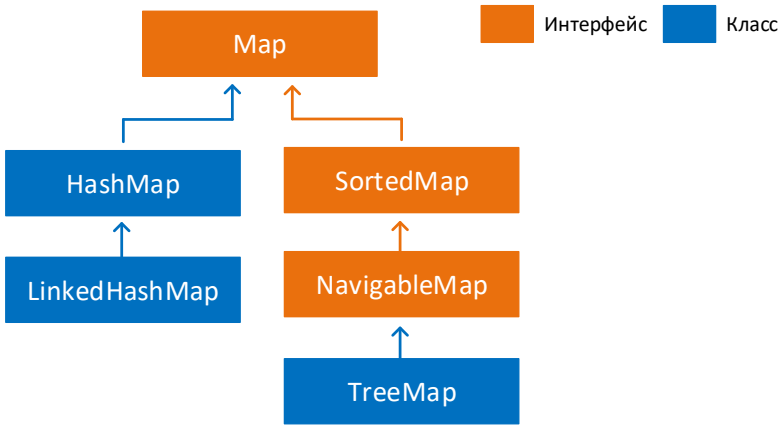
```
Измененный arrayList в обратном порядке: D+ C+ B+ A+
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorDemo {
    public static void main(String[] args) {
        List<String> arrayList = Arrays.asList("A", "B", "C", "D");
        ListIterator<String> listIterator = arrayList.listIterator();
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            listIterator.set(element + "+");
        }
        System.out.print("Измененный arrayList в обратном порядке: ");
        while (listIterator.hasPrevious()) {
            String element = listIterator.previous();
            System.out.print(element + " ");
        }
    }
}
```

Отображения Map

Отображение (или карта) представляет собой объект, сохраняющий связи между ключами и значениями в виде пар "ключ-значение". По заданному ключу можно найти его значение.

Ключи и значения являются объектами. Ключи должны быть уникальными, а значения могут быть дублированными.



В одних отображениях допускаются null ключи и null значения, а в других – они не допускаются.

Уникальность ключей определяет реализация метода equals().

Для корректной работы с картами необходимо переопределить методы equals() и hashCode(). Допускается добавление объектов без переопределения этих методов, но найти эти объекты в Map будет невозможно.

Интерфейс Map

Интерфейс Map отображает уникальные ключи на значения.

Ключ – это объект, который используется для последующего извлечения данных. Задавая ключ и значение, можно помещать значения в объект Map.

После того как это значение сохранено, можно получить его по ключу.

```
interface Map<K, V>
```

Здесь K указывает тип ключей, а V тип хранимых значений.

Методы интерфейса Map:

- void clear() – удаляет все пары "ключ-значение" из вызывающей карты.
- boolean containsKey(Object k) – возвращает true, если вызывающая карта содержит ключ k. В противном случае возвращает false.
- boolean containsValue(Object v) – возвращает true, если вызывающая карта содержит значение v. В противном случае возвращает false.
- Set<Map. Entry<K, V> entrySet() – возвращает Set, содержащий все значения карты. Набор содержит объекты типа Map.Entry. Т.е. этот метод представляет карту в виде набора.
- V get(Object K) – возвращает значение, ассоциированное с ключом k. Возвращает null, если ключ не найден.
- boolean isEmpty() – возвращает true, если вызывающая карта пуста. В противном случае возвращает false.
- Set<K> keySet() – возвращает Set, который содержит ключи вызывающей карты. Этот метод представляет ключи вызывающей карты в виде набора.
- V put(K k, V v) – помещает элемент в вызывающую карту, перезаписывая любое предшествующее значение, ассоциированное с ключом. Ключ и значение это k и v соответственно. Возвращает null, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом.
- void putAll(Map<? extends K, ? extends V> m) – помещает все значения из m в карту.
- V remove(Object k) – удаляет элемент, чей ключ равен k.
- int size() – возвращает число пар "ключ-значение" в карте.

- Collection<V> values() – возвращает коллекцию, содержащую значения карты. Этот метод представляет значения, содержащихся в карте, в виде коллекции.

Интерфейс SortedMap

Интерфейс SortedMap расширяет Map. Он гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

Методы интерфейса SortedMap:

- Comparator<? super K> comparator() – возвращает компаратор вызывающей сортированной карты. Если картой используется естественный порядок, возвращается null.
- K firstKey() – возвращает первый ключ вызывающей карты.
- K lastKey() – возвращает последний ключ вызывающей карты.
- SortedMap<K, V> headMap(K end) – возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше end.
- SortedMap<K, V> subMap(K start, K end) – возвращает карту, содержащую элементы вызывающей карты, чей ключ больше или равен start и меньше end.
- SortedMap<K, V> tailMap (K start) – возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых больше start.

Интерфейс NavigableMap

Интерфейс NavigableMap расширяет SortedMap и определяет поведение карты, поддерживающей извлечение элементов на основе ближайшего соответствия заданному ключу или ключам.

Методы интерфейса NavigableMap:

- Методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равную пару “ключ-значение” по отношению к заданному:
  - Map.Entry<K, V> lowerEntry(K key).
  - Map.Entry<K, V> floorEntry(K key).
  - Map.Entry<K, V> higherEntry(K key).
  - Map.Entry<K, V> ceilingEntry(K key).

- Методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равный ключ по отношению к заданному:
  - K lowerKey(K key).
  - K floorKey(K key).
  - K higherKey(K key).
  - K ceilingKey(K key).

- Методы pollFirstEntry и pollLastEntry возвращают соответственно первый и последний элементы карты, удаляя их из коллекции. Методы firstEntry и lastEntry также возвращают соответствующие элементы, но без удаления:
  - Map.Entry<K, V> pollFirstEntry().
  - Map.Entry<K, V> pollLastEntry().
  - Map.Entry<K, V> firstEntry().
  - Map.Entry<K, V> lastEntry().

- Возвращает карту, отсортированную в обратном порядке:
  - NavigableMap<K, V> descendingMap().

- Методы, позволяющие получить набор ключей, отсортированных в прямом и обратном порядке соответственно:
  - NavigableSet<K> navigableKeySet().

- NavigableSet<K> descendingKeySet().

- Методы, позволяющие извлечь из карты подмножество. Как и в случае NavigableSet, необходимо указать в параметрах начальный и конечный элементы массива ключей, а также необходимость включения в выходной набор граничных элементов. Опять же, если не указывать флаги, то будет использован интервал ключевых значений со включенным начальным элементом, но с исключенным конечным элементом:
  - NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive).
  - NavigableMap<K, V> headMap(K toKey, boolean inclusive).
  - NavigableMap<K, V> tailMap(K fromKey, boolean inclusive).
  - SortedMap<K, V> subMap(K fromKey, K toKey).
  - SortedMap<K, V> headMap(K toKey).
  - SortedMap<K, V> tailMap(K fromKey).

Интерфейс Map.Entry

Интерфейс Map.Entry позволяет работать с элементом карты. Метод entrySet(), объявленный в интерфейсе Map, возвращает Set, содержащий элементы карты. Каждый из элементов этого набора представляет собой объект типа Map.Entry.

Класс HashMap

Класс HashMap реализует интерфейс Map. Он использует хеш-таблицу для хранения карты. Это позволяет обеспечить константное время выполнения методов get() и put() даже при больших наборах.

Ключи и значения могут быть любых типов, в том числе и null.

HashMap обобщенный класс со следующим объявлением.

class HashMap<K, V>

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class HashMapDemo {
    public static void main(String[] args) {
        Map<String, Double> hashMap = new HashMap<>();
        hashMap.put("Иванов", 3434.34);
        hashMap.put("Петров", 123.22);
        hashMap.put("Сидоров", 1378.00);
        Set<String> keys = hashMap.keySet();
        for (String key : keys) {
            System.out.print(key + ": ");
            System.out.println(hashMap.get(key));
        }
    }
}
```

```
Иванов: 3434.34
Сидоров: 1378.0
Петров: 123.22
```

Класс TreeMap

TreeMap – хранит элементы в порядке сортировки. TreeMap сортирует элементы по возрастанию от первого к последнему. Порядок сортировки может задаваться реализацией интерфейсов Comparator и Comparable. Реализация Comparator передается в конструктор TreeMap, Comparable используется при добавлении элемента в карту.

Конструкторы класса TreeMap:

- TreeMap().

- TreeMap(Comparator<? super K> сотр).
- TreeMap(Map<? extends K, ? extends V> т).
- TreeMap(SortedMap<K, ? extends V> sm).

```
import java.util.SortedMap;
import java.util.TreeMap;

public class TreeMapDemo {
    public static void main(String[] args) {
        SortedMap<String, Double> treeMap = new TreeMap<>();
        treeMap.put("Иванов", 3434.34);
        treeMap.put("Петров", 123.22);
        treeMap.put("Сидоров", 1378.00);
        treeMap.forEach( action: (k, v) -> System.out.println(k + ": " + v));
    }
}
```

```
Иванов: 3434.34
Петров: 123.22
Сидоров: 1378.0
```

Класс LinkedHashMap

Класс LinkedHashMap расширяет HashMap. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать итерацию по карте в порядке вставки.

Класс Collections

Каркас коллекций определяет несколько алгоритмов, которые могут быть применимы к коллекциям и картам. Эти алгоритмы определены как статические методы в классе Collections.

Метод Collections.sort()

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class SortCollections {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый");
        System.out.println("Перед сортировкой: " + list);
        Collections.sort(list); // Сортирует элементы списка list в естественном порядке
        System.out.println("После сортировки: " + list);
        Collections.sort(list, c: Collections.reverseOrder()); // Возвращает обратный компаратор
        System.out.println("После обратной сортировки: " + list);
    }
}
```

```
Перед сортировкой: [красный, синий, зеленый]
После сортировки: [зеленый, красный, синий]
После обратной сортировки: [синий, красный, зеленый]
```

Метод Collections.binarySearch()

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class BinarySearchDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый");
        Collections.sort(list);
        System.out.println(list);
        // Отыскивает объект в списке list. Список должен быть сортированным.
        // Возвращает позицию объекта в слиске list или -, если значение не найдено
        System.out.println(Collections.binarySearch(list, key: "красный"));
        System.out.println(Collections.binarySearch(list, key: "черный"));
    }
}
```

```
[зеленый, красный, синий]
1
-4
```

Методы Collections.reverse(), Collections.shuffle()

```
Перед reversing: [красный, синий, зеленый, черный]
После reversing: [черный, зеленый, синий, красный]
После shuffling: [красный, зеленый, черный, синий]
```

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый", "черный");
        System.out.println("Перед reversing: " + list);
        Collections.reverse(list); // Реверсирует последовательность списка list
        System.out.println("После reversing: " + list);
        Collections.shuffle(list); // Перетасовывает (т.е. рандомизирует) элементы в списке list
        System.out.println("После shuffling: " + list);
    }
}
```

Метод Collections.fill()

```
[черный, черный, черный]
```

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsFillDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый");
        Collections.fill(list, obj: "черный"); // Присваивает объект каждому элементу списка list
        System.out.println(list);
    }
}
```

Методы Collections.max(), Collections.min()

```
9
2
```

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsMinMaxDemo {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(2, 2, 5, 8, 9);
        System.out.println(Collections.max( coll: list)); // Возвращает максимальный элемент коллекции
        System.out.println(Collections.min( coll: list)); // Возвращает минимальный элемент коллекции
    }
}
```

Метод Collections.copy()

```
[1, 2, 3, 7]
```

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsCopyDemo {
    public static void main(String[] args) {
        List<Integer> src = Arrays.asList(1, 2, 3);
        List<Integer> dest = Arrays.asList(4, 5, 6, 7);
        Collections.copy(dest, src); // Копирует элементы src в dest
        System.out.println(dest);
    }
}
```

Метод Collections.rotate()

```
[a, b, c, d, e]
[d, e, a, b, c]
[e, a, b, c, d]
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsRotateDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("a", "b", "c", "d", "e");
        System.out.println(list);

        Collections.rotate(list, distance: 2); // Сдвигает элементы list на заданное количество
        System.out.println(list);

        Collections.rotate(list, distance: -1); // Сдвигает элементы list на заданное количество
        System.out.println(list);
    }
}
```

Метод Collections.checkedCollection()

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;

public class MyCheckedCollection {
    public static void main(String[] a) {
        List myList = new ArrayList();
        myList.add("one");
        myList.add("two");
        myList.add("three");
        myList.add("four");
        // Используется для получения динамически типизированного представления указанной коллекции
        Collection checkList = Collections.checkedCollection( c: myList, type: String.class);
        System.out.println("Checked list content: " + checkList);
        myList.add(10);
        checkList.add(10); // throws ClassCastException
    }
}
```

Метод Collections.frequency()

```
2
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Collection;

public class CollectionsFrequencyDemo {
    public static void main(String[] args) {
        Collection<String> collection = Arrays.asList("red", "cyan", "red");
        // Возвращает количество элементов в указанной коллекции, равное указанному объекту
        System.out.println(Collections.frequency( c: collection, o: "red"));
    }
}
```

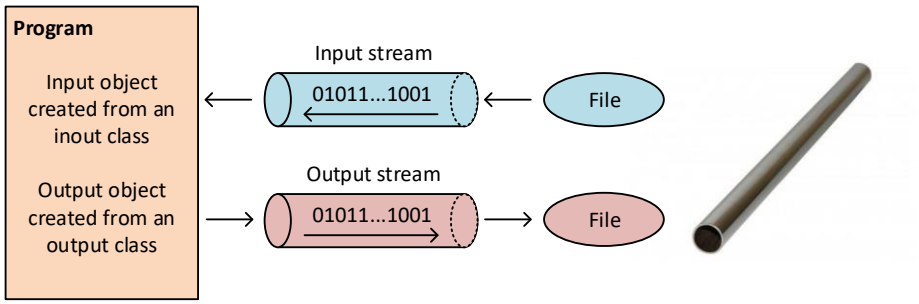
Задание

Необходимо создать класс Student, содержащий следующие характеристики – имя, группа, курс, оценки по предметам. Создать коллекцию, содержащую объекты класса Student. Написать метод, который удаляет студентов со средним баллом <4. Если средний балл >=4, студент переводится на следующий курс. Написать метод printStudents(List<Student> students, int course), который получает список студентов и номер курса, а также печатает на консоль имена тех студентов из списка, которые обучаются на данном курсе.

Потоки данных

В Java для описания работы по вводу/выводу используется специальное понятие – *поток данных* (stream). Поток данных связан с некоторым источником

или приемником данных, способных получать или предоставлять информацию. Соответственно, потоки делятся на входные – читающие данные, и на выходные – передающие (записывающие) данные.



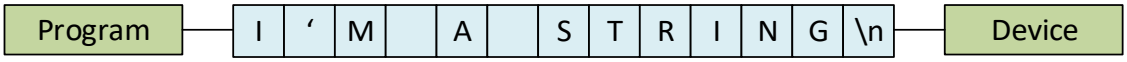
Потоки данных – это упорядоченные последовательности данных, которым соответствует определенный источник (source) (для потоков ввода) или получатель (destination) (для потоков вывода). В Java потоки ввода вывода реализуются в пределах иерархии классов, Определенных в пакете java.io. Классы ввода вывода Java исключают необходимость вникать в особенности низкоуровневой организации операционных систем и предоставляют доступ к системным ресурсам посредством методов работы с файлами и иных инструментов.

Все потоки ввода вывода ведут себя одинаково, несмотря на отличия в конкретных физических устройствах, с которыми они связаны. Одни и те же классы и методы ввода вывода применимы к разнотипным устройствам. Абстракция потока ввода может охватывать разные типы ввода: из файла на диске, клавиатуры или сетевого соединения.

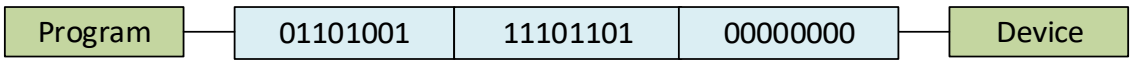
Типы потоков данных

В JAVA существует 2 типа потоков данных:

- Символьные потоки (text-streams, последовательности 16-битовых символов Unicode), содержащие символы.



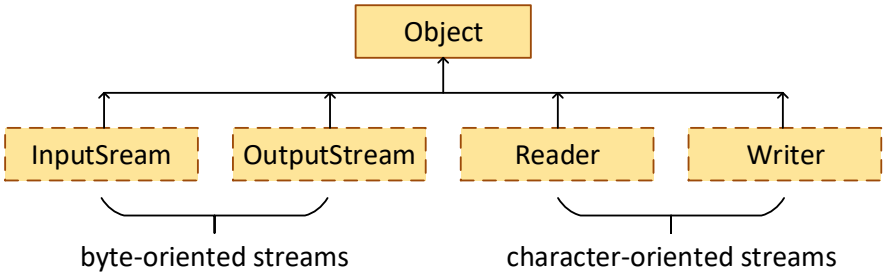
- Байтовые потоки (binary-streams), содержащие восьмибитную информацию.



Классы разделяются также по направлению потоков:

- Потоки ввода (input).
- Потоки вывода (output).

В Java определены четыре основных абстрактных классов для работы с потоками.



Общая схема работы с потоками в Java описывается тремя простыми шагами:

Шаг 1. Создать потоковый объект и ассоциировать его с файлом на диске.

Шаг 2. Пока есть информация, читать/писать очередные данные в/из потока.

Шаг 3. Закрыть поток.

Класс File

В отличие от большинства классов ввода/вывода, класс File работает не с потоками, а непосредственно с файлами. Данный класс позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

Конструкторы класса File:

- File(String path) – указывается путь к файлу.
- File(File dir, String name) – указывается объекта класса File (каталог) и имя файла.
- File(String dirPath, Sring name) – указывается путь к файлу и имя файла.
- File(URI uri) – указывается объекта URI, описывающий файл.

Методы класса File:

- getAbsolutePath() – абсолютный путь файла, начиная с корня системы.
- canRead() – доступно для чтения.
- canWrite() – доступно для записи.
- exists() – файл существует или нет.
- getName() – возвращает имя файла.
- getParent() – возвращает имя родительского каталога.
- getPath() – путь.
- lastModified() – дата последнего изменения.
- isFile() – объект является файлом, а не каталогом.
- isDirectory() – объект является каталогом.
- isAbsolute() – возвращает true, если файл имеет абсолютный путь.
- renameTo(File newPath) – переименовывает файл. В параметре указывается имя нового имени файла. Если переименование прошло неудачно, то возвращается false.
- delete() – удаляет файл. Также можно удалить пустой каталог.

Пример кода для получения информации о файле представлен ниже.

Каталог – это тоже объект класса File, который содержит список других файлов и каталогов. После создания объекта класса File, являющего каталогом, его метод isDirectory() вернет значение true.

Для создания каталога можно использовать метод mkdir(), который вернет true в случае успеха. Если указанный путь уже существует или каталог нельзя создать из-за отсутствия полного пути к нему, то вернется false.

Метод mkdirs() создает как сам каталог, так и всех его родителей.

Пример кода для получения содержимого каталога представлен ниже.

```java
import java.io.File;

public class FileDemo {
    public static void main(String[] args) {
        String catalogName = "src";
        File catalog = new File( pathname: catalogName);
        if (catalog.isDirectory()) {
            System.out.println("Папка " + catalogName);
            String[] list = catalog.list();
            if (list != null) {
                for (String fileName : list) {
                    File file = new File( pathname: catalogName + "/" + fileName);
                    if (file.isDirectory()) {
                        System.out.printf("\t%s каталог%n", fileName);
                    } else {
                        System.out.printf("\t%s файл%n", fileName);
                    }
                }
            }
        } else {
            System.out.println(catalogName + " не является каталогом");
        }
    }
}
```

Байтовые потоки

Класс InputStream

Класс InputStream – это абстрактный класс. Все байтовые потоки чтения наследуются от класса InputStream.

Методы класса InputStream:

- int read() – возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение –1.
- int read(byte[] buffer) – пытается прочесть максимум buffer.length байт из входного потока в массив buffer. Возвращает количество байт, в действительности прочитанных из потока. По достижении конца файла возвращает значение –1.
- int read(byte[] buffer, int offset, int length) – пытается прочесть максимум length байт, расположив их в массиве buffer, начиная с элемента offset. Возвращает количество реально прочитанных байт. По достижении конца файла возвращает –1. Методы read() будут блокированы, пока доступные данные не будут прочитаны:
- int available() – возвращает количество байтов ввода, доступные в данный момент для чтения.
- close() – закрывает источник ввода. Следующие попытки чтения передадут исключение IOException.
- long skip(long byteCount) – пропускает byteCount байт ввода, возвращая количество проигнорированных байтов.

Все методы выбрасывают исключение IOException, если происходит ошибка ввода вывода.

Класс OutputStream

Все байтовые потоки записи наследуются от абстрактного класса OutputStream.

Методы класса OutputStream:

- void write(int data) – записывает один байт в выходной поток. Аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.
- void write(byte[] buffer) – записывает в выходной поток весь указанный массив байт.
- void write(byte[] buffer, int offset, int length) – записывает в поток часть массива – length байт, начиная с элемента buffer[offset].
- flush() – очищает любые выходные буферы, завершая операцию вывода.
- close() – закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

Методы выбрасывают исключение IOException, если происходит ошибка ввода вывода.

Класс FileInputStream

Поток FileInputStream используется в Java для чтения данных из файла.

В примере ниже конструктор использует имя файла в качестве потока с целью создания объекта входного потока для считывания файла.

```
InputStream a = new FileInputStream( name: "D:/myprogramm/java/test.txt");
```

В следующем примере конструктор использует объектный файл с целью создания объекта входного потока для чтения файла.

```
File file = new File( pathname: "D:/myprogramm/java/test.txt");
InputStream a = new FileInputStream(file);
```

Класс FileOutputStream

В Java FileOutputStream используется для создания файла и последующей записи в него. Поток создаст файл в случае его отсутствия перед его открытием для вывода. Все методы выбрасывают исключение IOException, если происходит ошибка ввода вывода.

В примере ниже конструктор использует имя файла в качестве строки с целью создания объекта входного потока для записи файла в Java.

```
OutputStream b = new FileOutputStream( name: "D:/myprogramm/java/test.txt");
```

В следующем примере конструктор использует объектный файл с целью создания объекта выходного потока для записи файла.

```
File file = new File( pathname: "D:/myprogramm/java/test.txt");
OutputStream b = new FileOutputStream(file);
```

В программах ниже приведены примеры использования FileInputStream и FileOutputStream.

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

public class FileCopy {
    public static void main(String[] args) throws IOException {
        FileInputStream fileIn = null;
        FileOutputStream fileOut = null;

        try {
            fileIn = new FileInputStream( name: "src\\io\\file.txt");
            fileOut = new FileOutputStream( name: "src\\io\\copied_file.txt");

            int a;
            while ((a = fileIn.read()) != -1) {
                fileOut.write( b: a);
            }
        } finally {
            if (fileIn != null) {
                fileIn.close();
            }
            if (fileOut != null) {
                fileOut.close();
            }
        }
    }
}
```

Имя	Дата изменения	Тип	Размер
copied_file	03.10.2022 21:47	Текстовый докум...	1 КБ
file	03.10.2022 21:47	Текстовый докум...	1 КБ

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

public class FileInputOutputStreamDemo {
    public static void main(String[] args) {
        try {
            char[] symbols = {'a', 'b', 'c'};
            OutputStream output = new FileOutputStream( name: "a.txt");
            for (int i = 0; i < symbols.length; i++) {
                // Запись каждого символа в текстовый файл
                output.write( b: symbols[i]);
            }
            output.close();

            InputStream input = new FileInputStream( name: "a.txt");
            int size = input.available();

            for (int i = 0; i < size; i++) {
                // Чтение текстового файла посимвольно
                System.out.print((char) input.read() + " ");
            }
            input.close();
        } catch (IOException e) {
            System.out.print("Exception");
        }
    }
}
```

a b c

Имя	Дата изменения	Тип	Размер
.idea	03.10.2022 21:52	Папка с файлами	
out	01.10.2022 10:01	Папка с файлами	
src	03.10.2022 21:52	Папка с файлами	
a	03.10.2022 21:52	Текстовый докум...	1 КБ

Классы символьных потоков

Класс Reader

Класс Reader – абстрактный класс, определяющий символьный потоковый ввод. В случае ошибок все методы класса передают исключение IOException.

Методы класса Reader:

- int read() – возвращает представление очередного доступного символа во входном потоке в виде целого числа.
- int read(char[] buffer) – пытается прочесть максимум buffer.length символов из входного потока в массив buffer. Возвращает количество символов, в действительности прочитанных из потока.
- int read(char[] buffer, int offset, int length) – пытается прочесть максимум length символов, расположив их в массиве buffer, начиная с элемента offset. Возвращает количество реально прочитанных символов.

- close() – метод закрывает поток.

Класс Writer

Класс Writer – абстрактный класс, определяющий символьный потоковый вывод. В случае ошибок все методы класса передают исключение IOException.

Методы класса Writer:

- void write(int c) – записывает один символ в поток.
- void write(char[] buffer) – записывает массив символов в поток.
- void write(char[] buffer, int offset, int length) – записывает в поток подмассив символов длиной length, начиная с позиции offset.
- void write(String aString) – записывает строку в поток.
- void write(String aString, int offset, int length) – записывает в поток подстроку символов длиной length, начиная с позиции offset.

Класс FileWriter

Класс FileWriter создает объект класса, производного от класса Writer, который можно применять для записи файла. Создание объекта не зависит от наличия файла, он будет создан в случае необходимости. Если файл существует, и он доступен только для чтения, то передается исключение IOException.

Класс FileReader

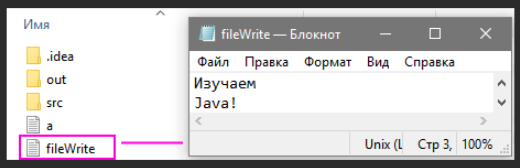
Класс FileReader, производный от класса Reader, можно использовать для чтения содержимого файла. В конструкторе класса нужно указать либо путь к файлу, либо объект типа File.

Все методы выбрасывают исключение IOException, если происходит ошибка ввода вывода.

В программах ниже приведены примеры использования FileWriter и FileReader.

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

public class FileWriterReader {
    public static void main(String[] args) {
        char[] array = new char[50];
        int size = 0;
        File file = new File( pathname: "fileWrite.txt");
        try (FileWriter fw = new FileWriter(file);
             FileReader fr = new FileReader(file)) {

            fw.write( str: "Изучаем\nJava!\n");
            fw.flush();

            size = fr.read( cbuf: array);
            System.out.println("Количество прочитанных символов: "
                    + size + " ");
            for (int i = 0; i < size; i++) {
                System.out.print(array[i]);
            }
        } catch (IOException e) {
            System.out.print(e.getMessage());
        }
    }
}
```

```
Изучаем
Java!
```



Класс BufferedWriter

Класс BufferedWriter записывает текст в поток, предварительно буферизируя записываемые символы, тем самым снижая количество обращений к физическому носителю для записи данных.

По сравнению с классом FileWriter, BufferedWriter записывает относительно большие куски данных в файл, минимизируя количество обращений к файлу. Другими словами, операции проходят быстрее. Добавляет метод void newLine().

Конструкторы класса BufferedWriter:

- BufferedWriter(Writer out).
- BufferedWriter(Writer out, int sz).

В качестве параметра он принимает поток вывода, в который надо осуществить запись. Второй параметр указывает на размер буфера.

В программе ниже приведен пример использования BufferedWriter.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedWriterDemo {
    public static void main(String[] args) {
        try (BufferedWriter bufferedWriter = new BufferedWriter(
                out: new FileWriter( fileName: "src\\io\\buffWriter.txt"))) {
            String text = "Привет мир!";
            bufferedWriter.write( str: text);
            bufferedWriter.newLine();
            bufferedWriter.write( str: text);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

buffWriter — Бл...
Файл Правка Формат Вид Справка
Привет мир!
Windo Стр 2, 1009

Класс BufferedReader

Класс BufferedReader считывает текст из символьного потока ввода, буферизируя прочитанные символы. Использование буфера призвано увеличить производительность чтения данных из потока. Добавляет метод String readLine().

Конструкторы класса BufferedReader:

- BufferedReader(Reader in);
- BufferedReader(Reader in, int sz).

Второй конструктор определяет потока ввода для чтения, а также размер буфера, в который будут считываться символы.

В программе ниже приведен пример использования BufferedReader.

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

public class BufferedReaderDemo {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(
                in: new FileReader( fileName: "src\\io\\file.txt"))) {
            String str;
            while ((str = br.readLine()) != null) {
                System.out.println(str);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

FileReader

file — Блокнот
Файл Правка Формат Вид Справка
FileReader
Windo Стр 1, 1009

Класс PrintWriter

Класс PrintWriter расширяет класс Writer. Содержит методы для форматирования.

Конструкторы класса PrintWriter:

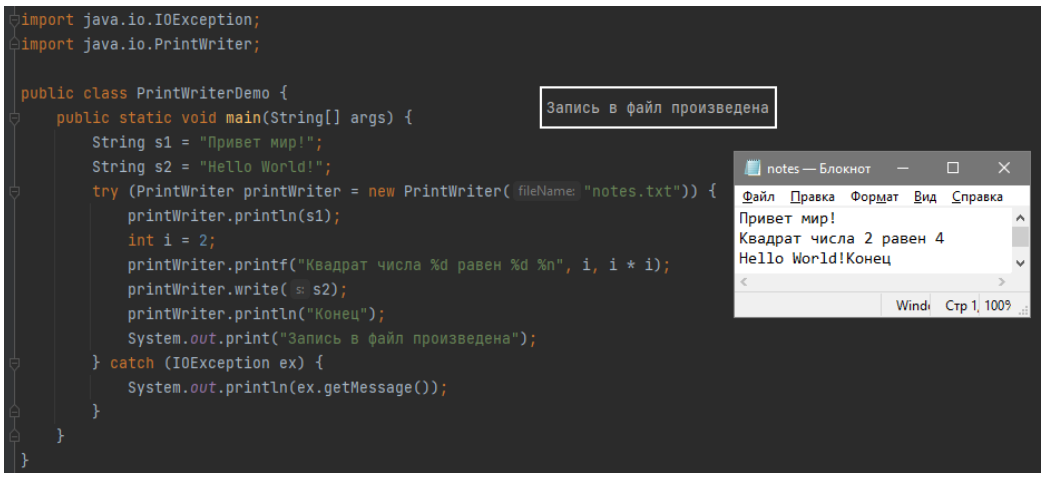
- PrintWriter(File file) – автоматически добавляет информацию в указанный файл;

- PrintWriter(String fileName) – автоматически добавляет информацию в файл по указанному имени;
- PrintWriter(OutputStream out) – для вывода информации используется существующий объект OutputStream, автоматически сбрасывая в него данные;
- PrintWriter(Writer out) – для вывода информации используется существующий объект Writer, в который автоматически идет запись данных.

Методы класса PrintWriter:

- println() – вывод строковой информации с переводом строки;
- print() – вывод строковой информации без перевода строки;
- printf() – форматированный вывод;
- format() – форматированный вывод.

В программе ниже приведен пример использования PrintWriter.



Задание

Необходимо создать класс, который копирует содержимое из одного файла в другой. Для этого нужно использовать классы BufferedReader, FileReader, BufferedWriter, FileWriter.

Javadoc

Наибольшая проблема, связанная с документированием кода – поддержка этой документации. Если документация и код разделены, возникают трудности, связанные с необходимостью внесения изменений в соответствующие разделы сопроводительной документации всякий раз при изменении программного кода. Среда разработки предлагает решение – связать код с документацией, поместив все в один файл.

Javadoc – генератор документации в HTML-формате из комментариев исходного кода на Java.

Комментарии документации применяют для документирования:

- классов,
- интерфейсов,
- полей (переменных),
- конструкторов,
- методов,
- пакетов.

В каждом случае комментарий должен находиться перед документируемым элементом. Документирование класса, метода или поля начинается с комбинации символов /**, после которого следует тело комментариев; заканчивается комбинацией символов */.

Утилита javadoc позволяет вставлять HTML теги и использовать специальные ярлыки (дескрипторы) документирования.

HTML теги заголовков не используют, чтобы не нарушать стиль файла, сформированного утилитой.

Дескрипторы javadoc, начинающиеся со знака @, называются автономными и должны помещаться с начала строки комментария (лидирующий символ * игнорируется).

Дескрипторы, начинающиеся с фигурной скобки, например {@code}, называются встроенными и могут применяться внутри описания.

Список дескрипторов Javadoc

Дескриптор	Применение	Описание
@author	Класс, интерфейс	Автор
@version	Класс, интерфейс	Версия. Не более одного дескриптора на класс
@since	Класс, интерфейс, поле, метод	Указывает, с какой версии доступно
@see	Класс, интерфейс, поле, метод	Ссылка на другое место в документации
@param	Метод	Входной параметр метода
@return	Метод	Описание возвращаемого значения
@exception имя_класса описание	Метод	Описание исключения, которое может быть послано из метода
@throws имя_класса описание	Метод	Описание исключения, которое может быть послано из метода
@deprecated	Класс, интерфейс, поле, метод	Описание устаревших блоков кода
{@link reference}	Класс, интерфейс, поле, метод	Ссылка
{@value}	Статичное поле	Описание значения переменной

Генерация файлов

Утилита javadoc в качестве входных данных принимает файл с исходным кодом программы. Генерирует несколько HTML файлов, содержащих документацию по этой программе. Информация о каждом классе будет содержаться в отдельном HTML файле. Кроме того, создается дерево индексов и иерархии. Могут быть сгенерированы и другие HTML файлы. Ctrl + Q – просмотр документации.

Сериализация, клонирование

Сериализация

Сериализация – это процесс сохранения состояния объекта в последовательность байт.

Десериализация – это процесс восстановления объекта, из этих байт.

Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов. Процесс сериализации заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора static или transient. Поля, помеченные ими, не могут быть предметом сериализации.

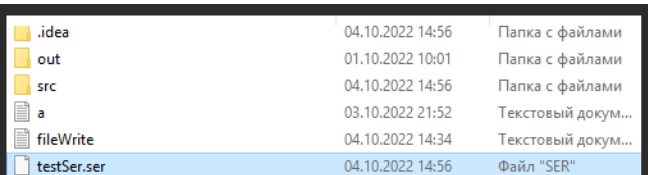
```java
import java.io.Serializable;

public class Cat implements Serializable {
    private String name;
    public Cat(String name) {
        this.name = name;
    }
}
```

Для того, чтобы объект мог быть сериализован, он должен реализовать интерфейс Serializable. Интерфейс java.io.Serializable не определяет никаких методов. Его присутствие только определяет, что объекты этого класса разрешено сериализовывать. При попытке сериализовать объект, не реализующий этот интерфейс, будет выброшено java.io.NotSerializableException.

После того как объект был сериализован (превращен в последовательность байт), его можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация).

При десериализации поле со спецификатором transient получает значение по умолчанию, соответствующее его типу.

Поле со спецификатором static не изменяется.




```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializeCat {
    private static final String FILE_NAME = "testSer.ser";

    public static void main(String[] args) {
        serialize();
        Cat cat = deserialize();
    }

    private static Cat deserialize() {
        Cat cat = null;
        try (FileInputStream fis = new FileInputStream(FILE_NAME);
             ObjectInputStream ois = new ObjectInputStream( in: fis)) {
            cat = (Cat) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
        }
        return cat;
    }

    private static void serialize() {
        Cat cat = new Cat( name: "Барсик");
        try (FileOutputStream fs = new FileOutputStream(FILE_NAME);
             ObjectOutputStream os = new ObjectOutputStream( out: fs)) {
            os.writeObject( obj: cat);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается.

Для работы с сериализацией в java.io определены интерфейсы ObjectInput, ObjectOutput и реализующие их классы ObjectInputStream и ObjectOutputStream соответственно. Для сериализации объекта нужен выходной поток OutputStream, который следует передать при конструировании ObjectOutputStream. После чего

вызовом метода `writeObject()` сериализовать объект и записать его в выходной поток.

Граф исходного объекта

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь так же могут хранить ссылки на другие объекты. И все ссылки тоже должны быть восстановлены при десериализации. Важно, что если несколько ссылок указывают на один и тот же объект, то в восстановленных объектах эти ссылки так же указывали на один и тот же объект. Чтобы сериализованный объект не был записан дважды, механизм сериализации некоторым образом для себя помечает, что объект уже записан в граф, и когда в очередной раз попадется ссылка на него, она будет указывать на уже сериализованный объект. Такой механизм необходим, чтобы иметь возможность записывать связанные объекты, которые могут иметь перекрестные ссылки. В таких случаях необходимо отслеживать, был ли объект уже сериализован, то есть нужно ли его записывать или достаточно указать ссылку на него.

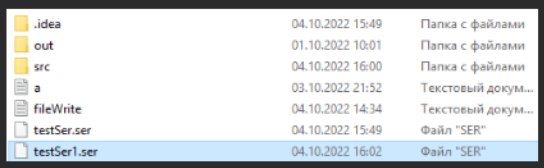
Если класс содержит в качестве полей другие объекты, то эти объекты так же будут сериализовываться и поэтому тоже должны быть сериализуемы. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и так далее. Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов, на которые у него имеются ссылки, и так далее – называется *графом исходного объекта*.

```java
import java.io.Serializable;

public class Dog implements Serializable {
    private Collar theCollar;
    public Dog(Collar collar) {
        theCollar = collar;
    }
    public Collar getCollar() {
        return theCollar;
    }
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Dog dog = (Dog) o;
        return theCollar != null ? theCollar.equals(dog.theCollar) : dog.theCollar == null;
    }
    public int hashCode() {
        return theCollar != null ? theCollar.hashCode() : 0;
    }
}
```

```java
import java.io.Serializable;

public class Collar implements Serializable {
    private int collarSize;
    public Collar(int size) {
        collarSize = size;
    }
    public int getCollarSize() {
        return collarSize;
    }
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Collar collar = (Collar) o;
        return getCollarSize() == collar.getCollarSize();
    }
    public int hashCode() {
        return getCollarSize();
    }
}
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SerializeDog {
    private static final String FILE_NAME = "testSer1.ser";
    public static void main(String[] args) {
        Collar collar = new Collar( size: 3);
        Dog dog1 = new Dog(collar);
        Dog dog2 = new Dog(collar);
        serialize( ...dogs: dog1, dog2);
        Dog[] deserializedDogs = deserialize( dogNumber: 2);
        System.out.println(dog1.equals(deserializedDogs[0]));
        System.out.println(dog2.equals(deserializedDogs[1]));
        Collar desCollar1 = deserializedDogs[0].getCollar();
        Collar desCollar2 = deserializedDogs[1].getCollar();
        System.out.println(desCollar1.equals(desCollar2));
    }
    private static void serialize(Dog... dogs) {
        try (FileOutputStream fs = new FileOutputStream(FILE_NAME);
             ObjectOutputStream os = new ObjectOutputStream( out: fs)) {
            for (Dog dog : dogs) {
                os.writeObject( obj: dog);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
    private static Dog[] deserialize(int dogNumber) {
        Dog[] dogs = new Dog[dogNumber];
        try (FileInputStream fs = new FileInputStream(FILE_NAME);
             ObjectInputStream os = new ObjectInputStream( in: fs)) {
            for (int i = 0; i < dogNumber; i++) {
                dogs[i] = (Dog) os.readObject();
            }
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
        }
        return dogs;
    }
}
```

```
true
true
true
```



Ключевое слово transient

Если по какой-то причине класс не может реализовать интерфейс Serializable, переменная этого класса может быть объявлена как transient.

```
import java.io.Serializable;

public class Bird implements Serializable {
    private String name;
    private transient Ring ring;
    public Bird(String name, Ring ring) {
        this.name = name;
        this.ring = ring;
    }
    public Ring getRing() {
        return ring;
    }
}
```

```
public class Ring {
    private int size;
    public Ring(int size) {
        this.size = size;
    }
    public int getSize() {
        return size;
    }
}
```

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializeBird {
    private static final String FILE_NAME = "testSer2.ser";
    public static void main(String[] args) {
        Ring ring = new Ring( size: 5);
        Bird bird = new Bird( name: "pigeon", ring);
        System.out.println("Размер кольца перед сериализацией: "
                + bird.getRing().getSize());
        try (FileOutputStream fs = new FileOutputStream(FILE_NAME);
             ObjectOutputStream os = new ObjectOutputStream( out: fs)) {
            os.writeObject( obj: bird);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        try (FileInputStream fis = new FileInputStream( name: "testSer2.ser");
             ObjectInputStream ois = new ObjectInputStream( in: fis)) {
            bird = (Bird) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Размер кольца после сериализации: "
                + bird.getRing().getSize());
    }
}
```

```
Размер кольца перед сериализацией: 5
Exception in thread "main" java.lang.NullPointerException Cannot invoke "Ring.getSize()" because the return value of "Bird.getRing()" is null
    at SerializeBird.main(SerializeBird.java:30)
```

.idea	04.10.2022 16:08	Папка с файлами
out	01.10.2022 10:01	Папка с файлами
src	04.10.2022 16:12	Папка с файлами
a	03.10.2022 21:52	Текстовый докум...
fileWrite	04.10.2022 14:34	Текстовый докум...
testSer.ser	04.10.2022 16:08	Файл "SER"
testSer1.ser	04.10.2022 16:02	Файл "SER"
testSer2.ser	04.10.2022 16:12	Файл "SER"

Десериализация и наследование

При десериализации производного класса, наследуемого от несериализуемого класса, вызывается конструктор без параметров родительского несериализуемого класса. И если такого конструктора не будет – при десериализации возникнет ошибка java.io.InvalidClassException. Конструктор же дочернего объекта, того, который десериализуем, не вызывается.

В процессе десериализации, поля не сериализуемых классов (родительских классов, не реализующих интерфейс Serializable) инициируются вызовом конструктора без параметров. Такой конструктор должен быть доступен из сериализуемого их подкласса. Поля сериализуемого класса будут восстановлены из потока.

```java
import java.util.Objects;

public class Insect {
    private String name;
    public Insect(String type) {
        this.name = type;
    }
    public Insect() {
    }
    public String getName() {
        return name;
    }
    public void setName(String type) {
        this.name = type;
    }
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Insect insect = (Insect) o;
        return Objects.equals( a: name,  b: insect.name);
    }
    public int hashCode() {
        return Objects.hash( ...values: name);
    }
    public String toString() {
        return "Insect{" + "name='" + name + '\'' + '}';
    }
}
```

```java
import java.io.Serializable;
import java.util.Objects;

public class Bug extends Insect implements Serializable {
    private boolean fly;
    public Bug(String type, boolean fly) {
        super(type);
        this.fly = fly;
    }
    public boolean isFly() {
        return fly;
    }
    public void setFly(boolean fly) {
        this.fly = fly;
    }
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        if (!super.equals(o)) return false;
        Bug bug = (Bug) o;
        return fly == bug.fly;
    }
    public int hashCode() {
        return Objects.hash( ...values: super.hashCode(), fly);
    }
    public String toString() {
        return "Bug{" + "fly=" + fly + "} " + super.toString();
    }
}
```

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializeInsect {
    private static final String FILE_NAME = "testSer.ser";

    public static void main(String[] args) {
        Bug bug = new Bug( type: "Майский жук",  fly: true);
        System.out.println("До сериализации " + bug);
        try (FileOutputStream fs = new FileOutputStream(FILE_NAME);
             ObjectOutputStream os = new ObjectOutputStream( out: fs)) {
            os.writeObject( obj: bug);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        try (FileInputStream fis = new FileInputStream(FILE_NAME);
             ObjectInputStream ois = new ObjectInputStream( in: fis)) {
            bug = (Bug) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("После сериализации " + bug);
    }
}
```

```
До сериализации Bug{fly=true} Insect{name='Майский жук'}
После сериализации Bug{fly=true} Insect{name='null'}
```

Клонирование

Иногда необходимо на основе существующего объекта создать второй такой же – то есть создать его клон. Это процесс в Java называется *клонированием*.

Для клонирования объекта в Java можно воспользоваться тремя способами:

- Переопределение метода clone() и реализация интерфейса Cloneable().
- Использование конструктора копирования.
- Использовать для клонирования механизм сериализации.

Переопределение метода clone()

Класс Object определяет метод clone(), который создает копию объекта. Если необходимо, чтобы экземпляр класса можно было клонировать, нужно переопределить этот метод и реализовать интерфейс Cloneable. Интерфейс Clonable – это интерфейс маркер, он не содержит ни методов, ни переменных. Интерфейсы маркер просто определяют поведение классов.

Object.clone() выбрасывает исключение CloneNotSupportedException при попытке клонировать объект, не реализующий интерфейс Cloneable.

Метод clone() в родительском классе Object объявлен как protected, поэтому желательно переопределить его как public. Реализация по умолчанию метода Object.clone() выполняет неполное/поверхностное (shallow) копирование.

В программе ниже приведен пример поверхностного клонирования.

```java
public class Car implements Cloneable {
    private String name;
    private Driver driver;
    public Car(String name, Driver driver) {
        this.name = name;
        this.driver = driver;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Driver getDriver() {
        return driver;
    }
    public void setDriver(Driver driver) {
        this.driver = driver;
    }
    public Car clone() throws CloneNotSupportedException {
        return (Car) super.clone();
    }
}
```

```java
public class Driver implements Cloneable {
    private String name;
    private int age;
    public Driver(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Driver clone() throws CloneNotSupportedException {
        return (Driver) super.clone();
    }
}
```

```java
public class CloneCar {
    public static void main(String[] args) throws CloneNotSupportedException {
        Car car = new Car( name: "Грузовик",  driver: new Driver( name: "Енукидзе Гиви",  age: 19));
        Car clonedCar = car.clone();
        System.out.println("Оригинал:\t" + car);
        System.out.println("Клон:   \t" + clonedCar);
        Driver clonedCarDriver = clonedCar.getDriver();
        clonedCarDriver.setName("Юрий Жуковец");
        System.out.println("Оригинал после изменения имени водителя:\t" + car);
        System.out.println("Клон после изменения имени водителя:   \t\t" + clonedCar);
    }
}
```

```
Оригинал:   Car@2d98a335
Клон:       Car@16b98e56
Оригинал после изменения имени водителя:    Car@2d98a335
Клон после изменения имени водителя:        Car@16b98e56
```

В этом примере клонируются объект класса Car. Клонирование выполняется поверхностное – новый объект clonedCar содержит ссылку на тот же объект

класса Driver, что и объект car. Для клонирования объекта необходимо написать "глубокое" клонирование – создать новый объект класса Driver.

Перепишем метод clone() класса Car.

```
    public Car clone() throws CloneNotSupportedException {
        Car newCar = (Car) super.clone();
        Driver driver = this.getDriver().clone();
        newCar.setDriver(driver);
        return newCar;
    }
}
```

Конструктор копирования

Еще один вариант клонирования объекта – это конструктор копирования. Создается конструктор, принимающий на вход объект того же класса, который необходимо клонировать.

В программе ниже приведен пример "неглубокого" клонирования, через конструктор копирования.

```
public class Driver implements Cloneable {
    private String name;
    private int age;
    public Driver(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Driver clone() throws CloneNotSupportedException {
        return (Driver) super.clone();
    }
}
```

Для реализации "глубокого" копирования необходимо переписать конструктор.

```
    public Car(Car otherCar) throws CloneNotSupportedException {
        this(otherCar.getName(), otherCar.getDriver().clone());
    }
```

Многопоточность

Многозадачность

Многозадачность бывает двух типов:

- на основе процессов;
- на основе потоков.

Многозадачность на основе процессов – это средство, которое позволяет одновременно выполнять две или несколько программ на компьютере. Процесс, по существу, является выполняющейся программой. Процессы являются крупными задачами, каждой из которых требуется свое адресное пространство. Связь между процессами ограничена и обходится дорого. Переключение контекста с одного процесса на другой также обходится дорого. Процесс может иметь несколько потоков. Количество потоков может меняться во время выполнения процесса.

Многозадачность на основе потоков – означает, что одна программа может выполнять две или несколько задач одновременно. Потоки исполнения более просты чем процессы. Они совместно используют одно и то же адресное пространство, и один и тот же крупный процесс. Связь между потоками исполнения обходится недорого, как, впрочем, и переключение контекста с одного потока исполнения на другой.

Многопоточность в Java

В Java многозадачность реализована на основе потоков. Многопоточное программирование применяют для сведения к минимуму времени простоя системы, поскольку сразу несколько задач могут выполняться одновременно. В Java поток определяет два разных понятия – "поток выполнения" и "объект класса java.lang.Thread".

При запуске Java-программы начинает выполняться главный поток (main). Особенность главного потока состоит в том, что в нем порождаются все дочерние потоки. Главный поток отождествляется с программой. Программа начинается с выполнения главного потока и должна завершаться с завершением главного потока. В отличие от дочерних потоков главный поток создается автоматически.

Потоки совместно используют:

- переменные класса и статические переменные;
- динамически распределяемую память (кучу);
- системные ресурсы, выделенные процессу.

Каждый поток имеет свои собственные локальные переменные (т.е. свой собственный стек).

Класс Thread

Класс Thread используется для создания нового потока.

В примере ниже продемонстрировано создание нового потока.

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("Все самое важное выполняется в потоке " + getName());
    }
}
```

```
public class MyThreadDemo {
        public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
    }
}
```

Метод start() – это специальный метод, создающий новый поток, в котором будет выполняться метод run(). Метод start() не может быть вызван дважды для одного и того же объекта, даже если поток завершен. Иначе будет выброшено и исключение IllegalThreadStateException.

Можно перегрузить метод run(), но он будет проигнорирован объектом Thread, если не вызвать его явно.

Конструкторы класса Thread:

- Thread();
- Thread(Runnable target);
- Thread(Runnable target, String name);
- Thread(String name).

В примере ниже показана реализация интерфейса java.lang.Runnable.

```java
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Все самое важное выполняется в потоке");
    }
}
```

```java
public class MyRunnableDemo {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();

        Thread thread1 = new Thread( target: myRunnable);
        Thread thread2 = new Thread( target: myRunnable);
        Thread thread3 = new Thread( target: myRunnable);

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

```
Все самое важное выполняется в потоке
Все самое важное выполняется в потоке
Все самое важное выполняется в потоке
```

Потоком можно управлять через объект Thread. Чтобы делать это, нужно получить ссылку на него вызовом метода currentThread(), который является общедоступным статическим (public static) методом класса Thread.

Пример получения имени текущего потока приведен ниже.

```java
public class NameRunnable implements Runnable {
    public void run() {
        System.out.println("NameRunnable running");
        System.out.println("Выполняется "
                + Thread.currentThread().getName());
    }
}
```

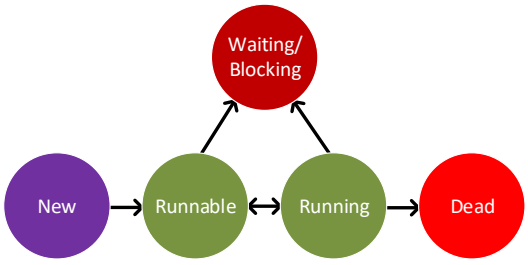
```java
public class NameThreadDemo {
    public static void main(String[] args) {
        NameRunnable nr = new NameRunnable();
        Thread tread1 = new Thread( target: nr);
        tread1.setName("Первый поток");
        tread1.start();

        Thread tread2 = new Thread( target: nr,  name: "Второй поток");
        tread2.start();
    }
}
```

```
NameRunnable running
NameRunnable running
Выполняется Первый поток
Выполняется Второй поток
```

Состояния потоков

Поток может находиться в одном из следующих состояний:

- New – объект класса Thread создан, но еще не запущен. Он еще не является потоком выполнения и естественно не выполняется.
- Runnable – поток готов к выполнению, но планировщик еще не выбрал его.
- Running – поток выполняется.
- Waiting/Blocked/Sleeping – поток блокирован или поток ждет окончания работы другого потока.
- Dead – поток завершен. Будет выброшено исключение при попытке вызвать метод start() для dead потока.



Существуют перечисление Thread.State, содержащее значения возможных состояний потока: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.

Получить текущее значение состояния потока можно вызовом метода getState() класса Thread.

Переключение между потоками

Планировщик потоков – это часть JVM, которая решает какой поток должен выполнится в каждый конкретный момент времени и какой поток нужно приостановить.

Действия при переключении контекста:

- Сохранение контекста потока, закончившего выполнение.
- Помещение этого потока в конец очереди, соответствующей его приоритету.
- Загрузка контекста потока из очереди готовых к выполнению с наибольшим приоритетом.
- Удаление загруженного потока из очереди и начало его выполнениясь.

Правила, определяющие, когда должно происходить переключение контекста:

- Поток может переключится при окончании выделенного планировщиком кванта времени.
- Поток может добровольно уступить управление. Для этого достаточно явно уступить очередь на исполнение, приостановить или блокировать поток на время ожидания ввода вывода.
- Один поток исполнения может быть вытеснен другим, более приоритетным потоком.

Когда речь идет о потоках, Java практически ничего не гарантирует.

Java не гарантирует, что потоки будут выполнены в том порядке в котором они были запущены. Нет гарантии, что если поток начал свое выполнение, то он выполнит свою работу не прерываясь. При каждом новом запуске программы, результат может быть разным. Гарантируется, что каждый поток начнет свою работу и будет выполнять пока не закончит. Внутри одного потока действия выполняются в предсказуемом порядке.

Методы, позволяющие влиять на планировщика потоков:

- Методы класса java.lang.Thread:
  - public static void sleep(long millis) throws InterruptedException;
  - public static void yield();
  - public final void join() throws InterruptedException;
  - public final void setPriority(int newPriority).
- Методы класса java.lang.Object:
  - public final void wait() throws InterruptedException;
  - public final void notify();
  - public final void notifyAll().

Метод Thread.sleep()

Можно приостановить выполнение потока на заданное время с помощью статического метода Thread.sleep(). Это бывает необходимо, когда поток выполняется слишком быстро, и когда нужно переключиться на другой поток.

В примере ниже показано использование метода Thread.sleep().

```java
public class SleepRunnable implements Runnable {
    public void run() {
        for (int x = 1; x < 4; x++) {
            System.out.println("Выполняется "
                    + Thread.currentThread().getName()
                    + ", x из " + x);
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

```java
public class SleepRunnableDemo {
    public static void main(String[] args) {
        SleepRunnable sleepRunnable = new SleepRunnable();

        Thread one = new Thread( target: sleepRunnable);
        one.setName("Костя");
        Thread two = new Thread( target: sleepRunnable);
        two.setName("Маша");
        Thread three = new Thread( target: sleepRunnable);
        three.setName("Юра");

        one.start();
        two.start();
        three.start();
    }
}
```

```
Выполняется Юра, x из 1
Выполняется Маша, x из 1
Выполняется Костя, x из 1
Выполняется Костя, x из 2
Выполняется Маша, x из 2
Выполняется Юра, x из 2
Выполняется Маша, x из 3
Выполняется Костя, x из 3
Выполняется Юра, x из 3
```

Метод join()

Нестатический метод join() приостановит выполнение текущего потока до тех пор, пока другой поток не закончит свое выполнение.

void join(long millis) – этот метод приостановит выполнение текущего потока на указанное время в миллисекундах. Выполнение этого метода зависит от реализации ОС, поэтому Java не гарантирует, что текущий поток будет ждать указанное время.

В примере ниже показано использование метода join().

```java
public class JoinRunnable extends Thread {
    public JoinRunnable(String name) {
        super(name);
    }

    public void run() {
        String currentThreadName = Thread.currentThread().getName();
        for (int i = 0; i < 4; i++) {
            System.out.println(currentThreadName + " Работает!" + i);
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(currentThreadName + " Завершен");
    }
}
```

```java
public class JoinDemo {
    public static void main(String[] args) {
        JoinRunnable a = new JoinRunnable( name: "A");
        JoinRunnable b = new JoinRunnable( name: "B");
        JoinRunnable c = new JoinRunnable( name: "C");

        a.start();
        try {
            a.join();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        b.start();
        c.start();
    }
}
```

```
A Работает!0
A Работает!1
A Работает!2
A Работает!3
A Завершен
B Работает!0
C Работает!0
B Работает!1
C Работает!1
B Работает!2
C Работает!2
B Работает!3
C Работает!3
B Завершен
C Завершен
```

Метод isAlive()

Метод isAlive() позволяет выяснить, используется поток или нет.

Если поток создан, но не запущен, метод вернет false.

Как только метод start() вызван для потока, он считается alive и метод isAlive() вернет true.

В примере ниже показано использование метода isAlive().

```
public class IsAliveDemo {
    public static void main(String[] args) throws InterruptedException {
        MyRunnable myRunnable = new MyRunnable();

        Thread thread = new Thread( target: myRunnable);
        System.out.println("До запуска: " + thread.isAlive());
        thread.start();
        System.out.println("После запуска: " + thread.isAlive());
        thread.join();
        System.out.println("После завершения потока: " + thread.isAlive());
    }
}
```

```
До запуска: false
После запуска: true
Все самое важное выполняется в потоке
После завершения потока: false
```

Задание

Необходимо создать класс NewThread расширяющий Thread. Переопределить метод run(). В цикле for вывести на консоль символ 100 раз. Создать экземпляр класса и запустить новый поток.

Создать класс, реализующий интерфейс Runnable. Переопределить run() метод. Создать цикл for. В цикле распечатать значения от 0 до 100 делящиеся на 10 без остатка. Необходимо используем статический метод Thread.sleep(), чтобы сделать паузу. Создать три потока, выполняющих задачу распечатки значений.

Синхронизация потоков

Все потоки, принадлежащие одному процессу, разделяют некоторые общие ресурсы (адресное пространство, открытые файлы). Что произойдет, если один поток еще не закончил работать с каким-либо общим ресурсом, а система переключилась на другой поток, использующий тот же ресурс?

Когда два или более потоков имеют доступ к одному разделенному ресурсу, они нуждаются в обеспечении того, что ресурс будет использован только одним потоком одновременно. Процесс, с помощью которого это достигается, называется *синхронизацией*.

В примере ниже показан одновременный доступ к ресурсу.

```
public class AccountDanger{
    public static void main(String[] args) {
        Account account = new Account();
        Thread one = new Thread( target: account);
        Thread two = new Thread( target: account);
        one.setName("Yenukidze Givi");
        two.setName("Skornyakov Fedor");
        one.start();
        two.start();
    }
}
```

```
Yenukidze Givi is going to withdraw
Skornyakov Fedor is going to withdraw
Skornyakov Fedor completes the withdrawal. The balance is 30
Yenukidze Givi completes the withdrawal. The balance is 30
Skornyakov Fedor is going to withdraw
Yenukidze Givi is going to withdraw
Skornyakov Fedor completes the withdrawal. The balance is 20
Skornyakov Fedor is going to withdraw
Yenukidze Givi completes the withdrawal. The balance is 20
Yenukidze Givi is going to withdraw
Skornyakov Fedor completes the withdrawal. The balance is 10
Yenukidze Givi completes the withdrawal. The balance is 0
Not enough in account for Yenukidze Givi to withdraw 0
Not enough in account for Yenukidze Givi to withdraw 0
Not enough in account for Skornyakov Fedor to withdraw 0
Not enough in account for Skornyakov Fedor to withdraw 0
```

```java
public class Account implements Runnable {
    private int balance = 50;
    public int getBalance() {
        return balance;
    }
    public void withdraw(int amount) {
        balance -= amount;
    }
    public void run() {
        for (int x = 0; x < 5; x++) {
            makeWithdrawal( amount: 10);
            if (getBalance() < 0) {
                System.out.println("account is overdrawn!");
            }
        }
    }
    private void makeWithdrawal(int amount) {
        if (getBalance() >= amount) {
            System.out.println(Thread.currentThread().getName()
                    + " is going to withdraw");
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            withdraw(amount);
            System.out.println(Thread.currentThread().getName()
                    + " completes the withdrawal. The balance is "
                    + getBalance());
        } else {
            System.out.println("Not enough in account for "
                    + Thread.currentThread().getName()
                    + " to withdraw " + getBalance());
        }
    }
}
```

Здесь два потока находятся в состоянии гонок. *Состояние гонок* – это одновременный вызов в потоках исполнения одного и того же метода для того же самого объекта.

Чтобы защитить данные, необходимо выполнить два действия:

- Объявить переменные как private.
- Синхронизировать код.

Синхронизировать прикладной код можно двумя способами:

С помощью синхронизированных методов. Метод объявляется с использованием ключевого слова `synchronized`:

- `public synchronized void someMethod()`.

Заключить вызовы методов в блок оператора `synchronized`:

- `synchronized(obj)` { // операторы, подлежащие синхронизации }.

Только методы и блоки могут быть синхронизированы, но не переменные и классы.

Не все методы в классе должны быть синхронизированы.

Модификатор `volatile`

Поток создается с чистой рабочей памятью, и должен перед использованием загрузить все необходимые переменные из основного хранилища (можно сказать что он имеет некий кэш).

Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее применять.

Если переменная объявлена, как volatile, то ее чтение и запись будет производиться из\в основное хранилище.

Чтение volatile переменных синхронизировано и запись в volatile переменные синхронизирована, а неатомарные операции – нет.

Монитор

Каждый объект в Java имеет ассоциированный с ним монитор. *Монитор* – это объект, используемый в качестве взаимоисключающей блокировки. Когда поток исполнения запрашивает блокировку, то говорят, что он входит в монитор.

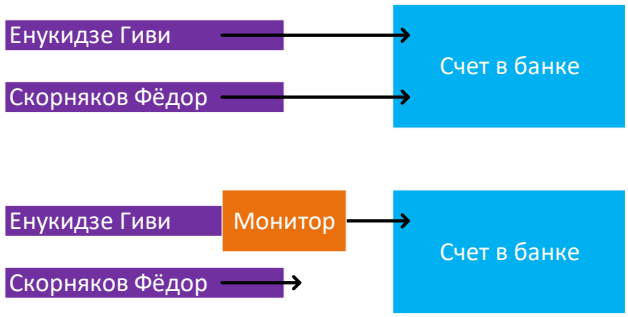
Только один поток исполнения может в одно, и то же время владеть монитором. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Говорят, что они ожидают монитор.

Поток, владеющий монитором, может, если пожелает, повторно войти в него.

Если поток засыпает, то он удерживает монитор.

Поток может захватить сразу несколько мониторов.

Рассмотрим разницу между доступом к объекту без синхронизации и из синхронизированного кода. На рисунке ниже приведены схемы доступа к банковскому счету без синхронизации и с синхронизацией.



Когда выполнение кода доходит до оператора synchronized, монитор объекта блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку (Енукидзе Гиви).

После окончания работы блока кода, монитор объекта освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

Синхронизируем методы из предыдущего примера для корректного совместного доступа потоков.

```java
public class AccountDanger implements Runnable {
    private Account account = new Account();
    public static void main(String[] args) {
        AccountDanger accountDanger = new AccountDanger();
        Thread one = new Thread( target: accountDanger);
        Thread two = new Thread( target: accountDanger);
        one.setName("Yenukidze Givi");
        two.setName("Skornyakov Fedor");
        one.start();
        two.start();
    }
    public void run() {
        for (int x = 0; x < 5; x++) {
            makeWithdrawal( amt: 10);
            if (account.getBalance() < 0) {
                System.out.println("account is overdrawn!");
            }
        }
    }
    private synchronized void makeWithdrawal(int amt) {
        if (account.getBalance() >= amt) {
            System.out.println(Thread.currentThread().getName()
                    + " is going to withdraw");
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            account.withdraw( amount: amt);
            System.out.println(Thread.currentThread().getName()
                    + " completes the withdrawal. The balance is "
                    + account.getBalance());
        } else {
            System.out.println("Not enough in account for "
                    + Thread.currentThread().getName()
                    + " to withdraw " + account.getBalance());
        }
    }
}
```

```
Yenukidze Givi is going to withdraw
Yenukidze Givi completes the withdrawal. The balance is 40
Yenukidze Givi is going to withdraw
Yenukidze Givi completes the withdrawal. The balance is 30
Yenukidze Givi is going to withdraw
Yenukidze Givi completes the withdrawal. The balance is 20
Yenukidze Givi is going to withdraw
Yenukidze Givi completes the withdrawal. The balance is 10
Skornyakov Fedor is going to withdraw
Skornyakov Fedor completes the withdrawal. The balance is 0
Not enough in account for Skornyakov Fedor to withdraw 0
Not enough in account for Skornyakov Fedor to withdraw 0
Not enough in account for Skornyakov Fedor to withdraw 0
Not enough in account for Skornyakov Fedor to withdraw 0
Not enough in account for Yenukidze Givi to withdraw 0
```

Синхронизация статических методов

Статические методы тоже могут быть синхронизированы с помощью ключевого слова synchronized.

Для синхронизации статических методов используется один монитор для одного класса. Каждый загруженный в Java класс имеет соответствующий объект класса Class, представляющий этот класс. Монитор именно этого объекта используется для синхронизации статических методов (если они синхронизированы).

```java
public static synchronized int getCount() {
    return count;
}
```

```java
public static int getCount() {
    synchronized(MyClass.class) {
        return count;
    }
}
```

Блокировка

Если поток пытается зайти в синхронизированный метод, а монитор уже захвачен, то поток блокируется по монитору объекта.

Поток попадает в специальный пул для этого конкретного объекта и должен находиться там пока монитор не будет освобожден. После этого поток возвращается в состояние runnable.

Варианты блокировки:

- Потоки, вызывающие нестатические синхронизированные методы одного и того же класса, будут блокировать друг друга только если они вызваны для одного объекта.
- Потоки, вызывающие статические синхронизированные методы одного класса, будут всегда блокировать друг друга. Они блокируются по монитору Class объекта. Статические синхронизированные и нестатические синхронизированные методы не будут блокировать друг друга никогда.
- Для синхронизированных блоков нужно смотреть какой объект используется для синхронизации.
- Блоки, синхронизированные по одному объекту, будут блокировать друг друга.

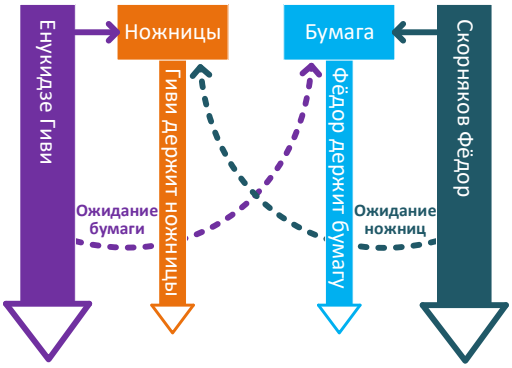
Задание

Необходимо создать три потока, которые изменяют один и тот же объект. Каждый поток должен вывести на экран одну букву 100 раз, и затем увеличить значение символа на 1. Создать класс, расширяющий Thread. Переопределить метод run() – в нем должен находиться синхронизированный блок кода. Для того чтобы три объекта потока имели доступ к одному объекту, необходимо создать конструктор, принимающий на вход StringBuilder объект. Синхронизированный блок кода должен получать монитор на объект StringBuilder созданный в конструкторе. Внутри синхронизированного блока кода вывести StringBuilder на экран 100 раз, а затем увеличить значение символа на 1. В методе main() создать один объект класса StringBuilder, используя символ 'a'. Затем создать три экземпляра объекта класса и запустить потоки.

Взаимная блокировка

Особый тип ошибок, которого следует избегать, имеющий отношение к многозадачности – это *взаимная блокировка* (deadlock). Она происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов.

Например, два студента Гиви и Фёдор провинились и делают стенд с фото в деканате. Для работы каждому нужны ножницы и бумага. Предположим Гиви взял ножницы (поток Гиви вошел в монитор объекта ножницы), а Фёдор бумагу (поток Фёдор вошел в монитор объекта бумага). Каждый из них ждет другой предмет и не хочет делиться тем, что взял. Они не могут продолжить свою работу и будут ждать вечно (пока зам. декана по воспитательной работе не поможет им).



В программе ниже приведен пример взаимной блокировки.

```java
public class DeadlockRisk implements Runnable {
    private static class Resource {
    }
    private final Resource scissors = new Resource();
    private final Resource paper = new Resource();
    public void doSun() {
        synchronized (scissors) { // May deadlock here
            System.out.println(Thread.currentThread().getName() + " взял ножницы для вырезания рамки");
            synchronized (paper) {
                System.out.println(Thread.currentThread().getName() + " взял бумагу для вырезания рамки");
                System.out.println(Thread.currentThread().getName() + " вырезает рамку");
            }
        }
    }
    public void doCloud() {
        synchronized (paper) { // May deadlock here
            System.out.println(Thread.currentThread().getName() + " взял бумагу для вырезания фото");
            synchronized (scissors) {
                System.out.println(Thread.currentThread().getName() + " взял ножницы для вырезания фото");
                System.out.println(Thread.currentThread().getName() + " вырезает фото");
            }
        }
    }
    public void run() {
        doSun();
        doCloud();
    }
    public static void main(String[] args) {
        DeadlockRisk job = new DeadlockRisk();
        Thread Givi = new Thread( target: job,  name: "Гиви");
        Thread Fedor = new Thread( target: job,  name: "Фёдор");
        Givi.start();
        Fedor.start();
    }
}
```

```
Гиви взял ножницы для вырезания рамки
Гиви взял бумагу для вырезания рамки
Гиви вырезает рамку
Фёдор взял ножницы для вырезания рамки
Фёдор взял бумагу для вырезания рамки
Фёдор вырезает рамку
Фёдор взял бумагу для вырезания фото
Фёдор взял ножницы для вырезания фото
Фёдор вырезает фото
Гиви взял бумагу для вырезания фото
Гиви взял ножницы для вырезания фото
Гиви вырезает фото
```

Межпотоковые коммуникации

В Java внедрен изящный механизм взаимодействия потоков исполнения с помощью методов wait(), notify() и notifyAll(). Эти методы реализованы как final в классе Object, поэтому они доступны всем классам.

Все три метода могут быть вызваны только из synchronized контекста.

Метод wait() принуждает вызывающий поток отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток не войдет в тот же монитор и не вызовет notify().

Метод notify() возобновляет работу потока, который вызвал wait() на том же самом объекте.

Метод notifyAll() возобновляет работу всех потоков, который вызвали wait() на том же самом объекте и одному из потоков дается доступ.

В коде ниже приведен пример взаимодействия потоков.

```java
public class MyQueue {
    private int n;
    boolean valueSet = false;

    public synchronized int get() {
        while (!valueSet) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Получено: " + n);
        valueSet = false;
        notify();
        return n;
    }

    public synchronized void put(int n) {
        while (valueSet) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        valueSet = true;
        this.n = n;
        System.out.println("Отправлено: " + n);
        notify();
    }
}
```

```java
public class Producer implements Runnable {
    private MyQueue myQueue;

    public Producer(MyQueue myQueue) {
        this.myQueue = myQueue;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            myQueue.put( n: i);
        }
    }
}
```

```java
public class Consumer implements Runnable {
    private MyQueue myQueue;

    public Consumer(MyQueue myQueue) {
        this.myQueue = myQueue;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            myQueue.get();
        }
    }
}
```

```java
public class ProducerDemo {
    public static void main(String[] args) {
        MyQueue myQueue = new MyQueue();

        Consumer consumer = new Consumer(myQueue);
        Producer producer = new Producer(myQueue);

        Thread t1 = new Thread( target: consumer);
        Thread t2 = new Thread( target: producer);

        t1.start();
        t2.start();
    }
}
```

```
Отправлено: 0
Получено: 0
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
...
```

Задание

Необходимо изменить класс MyQueue из примера выше. Вместо int n добавить Queue<T> (MyQueue сделать обобщенным), который будет содержать объекты, создаваемые Producer. Добавить еще один объект Consumer, который будет запускаться тоже отдельным потоком. Вывести на консоль какой из объектов Consumer обработал объект из очереди. Изменить цикл for на бесконечный.