ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 7. Важнейшие объектные типы

7.1. Массивы

Массив (*array*) – это упорядоченный набор одинаково устроенных ячеек, *доступ* к которым осуществляется по индексу. Например, если у массива имя `a1`, то `a1[i]` – имя ячейки этого массива, имеющей *индекс* `i`.

В *Java массивы являются объектами*, но особого рода – их объявление отличается от объявления других видов объектов. *Переменная* типа *массив* является ссылочной – в ней содержится *адрес* объекта, а не сам *объект*, как и для всех других *объектных переменных* в *Java*. В качестве элементов (ячеек) массива могут выступать значения как *примитивных типов*, так и *ссылочных типов*, в том числе – переменные типа *массив*.

Тип ячейки массива называется *базовым типом* для массива.

Для задания массива, в отличие от объектов других типов, не требуется предварительно задавать *класс*, и иметь специальное имя для данного *объектного типа*. Вместо имени класса при объявлении переменной используется имя базового типа, после которого идут пустые квадратные скобки.

Например, объявление

```
int[] a1;
```

задает переменную `a1` типа *массив*. При этом размер массива (число ячеек в нем) заранее не задается и не является частью типа.

Для того, чтобы создать *объект* типа *массив*, следует воспользоваться зарезервированным словом `new`, после чего указать имя базового типа, а за ним в квадратных скобках число ячеек в создаваемом массиве:

```
a1=new int[10];
```

Можно совместить объявление типа переменной и *создание массива*:

```
int[] a1=new int[10];
```

После *создания массивы Java* всегда инициализированы – в ячейках содержатся нули. Поэтому если базовый *тип массива* примитивный, элементы массива будут нулями соответствующего типа. А если *базовый тип* ссылочный – в ячейках будут значения `null`.

Ячейки в массиве имеют индексы, всегда начинающиеся с нуля. То есть первая *ячейка* имеет номер 0, вторая – номер 1, и так далее. Если *число элементов в массиве* равно `n`, то последняя *ячейка* имеет *индекс* `n-1`. Такая своеобразная *нумерация* принята в языках C и C++, и язык *Java* унаследовал эту не очень привлекательную особенность, часто приводящую к ошибкам при организации циклов.

Длина массива хранится в *поле* `length`, которое доступно только по чтению – изменять его путем присваивания нового значения нельзя.

Пример работы с массивом:

```
int[] a=new int[100];
for(int i=0;i<a.length;i++){
  a[i]=i+1;
};
```

Если у нас имеется *переменная* типа *массив*, и ей сопоставлен *массив* заданной длины, в любой момент этой переменной можно сопоставить новый *массив*. Например,

```
a1=new int[20];
```

При этом прежний *объект-массив*, находящийся в динамической области памяти, будет утерян и превратится в мусор.

Переменные типа *массив* можно присваивать друг другу. Например, если мы задали переменную

```
int[] a2;
```

то сначала в ней хранится *значение* `null` (*ссылка* направлена "в никуда"):

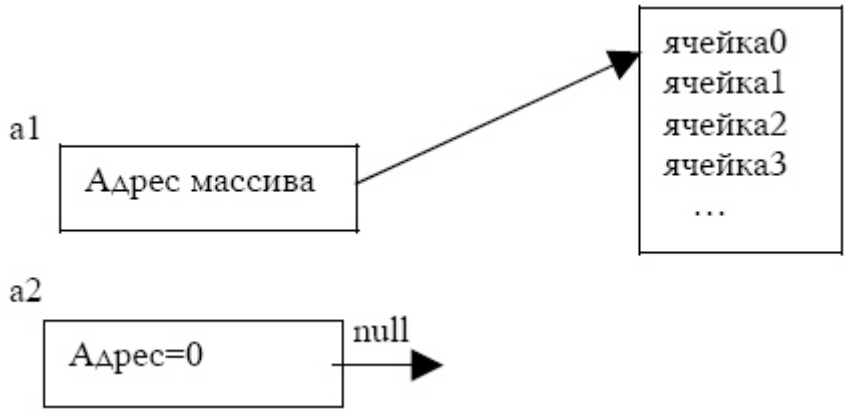

Рис. 7.1. Массив с ячейками типа int

Присваивание

```
a2=a1;
```

приведет к тому, что ссылочные переменные `a1` и `a2` будут ссылаться на один и тот же *массив*, расположенный в динамической области памяти.

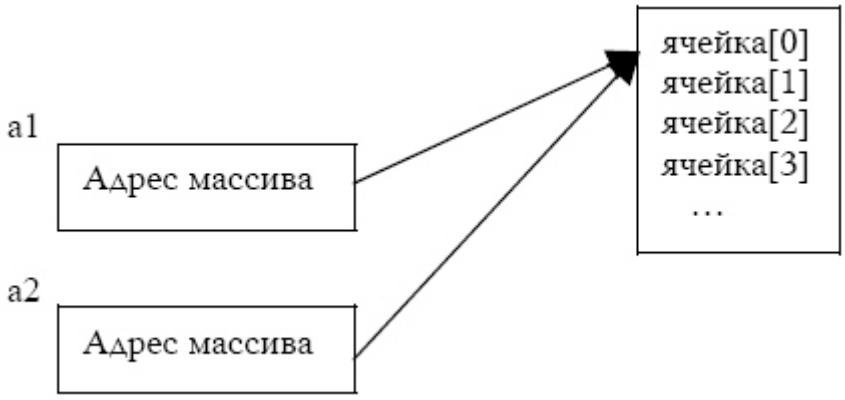

Рис. 7.2. Массив с ячейками типа int

То есть *присваивание* переменных типа *массив* приводит к тому, что имена переменных становятся синонимами одного и того же массива – копируется *адрес* массива. А вовсе не приводит к копированию элементов из одного массива в другой, как это происходит в некоторых других языках программирования.

В качестве элементов массивов могут выступать объекты. В этом случае *доступ* к полям и методам этих объектов производится через имя ячейки массива, после которого через точку указывается имя поля или метода. Например, если у нас имеется *класс* `Circle` ("*окружность*"), у которого имеются поля `x`, `y` и `r`, а также методы `show()` и `hide()`, то *массив* `circles` из 10 объектов такого типа может быть задан и инициализирован, например, так

```
int n=10;
Circle[] circles=new Circle[n];
for(int i=0;i<n;i++){
 circles[i]=new Circle();
 circles[i].x=40*i;
 circles[i].y= circles[i].x/2;
 circles[i].r=50;
 circles[i].show();
};
```

В такого рода программах для повышения читаемости часто применяется использование вспомогательной ссылки, позволяющей избежать многократного обращения по индексу. В нашем случае мы будем использовать

в этих целях переменную `circle`. На скорости работы программы это почти не сказывается (хотя и может чуть повысить *быстродействие*), но делает код более читаемым:

```
int n=10;
Circle[] circles=new Circle[n];
Circle circle;
for(int i=0;i<n;i++){
 circle=new Circle();
 circle.x=40*i;
 circle.y= circles[i].x/2;
 circle.r=50;
 circle.show();
 circles[i]= circle;
};
```

С помощью переменной `circle` мы инициализируем создаваемые объекты и показываем их на экране, после чего присваиваем ссылку на них ячейкам массива.

Двумерный массив представляет собой *массив* ячеек, каждая из которых имеет тип "*одномерный массив*". Соответствующим образом он и задается. Например, задание двумерного массива целых чисел будет выглядеть так:

```
int[][] a=new int[10][20];
```

Будет задана *ячейка* типа "*двумерный массив*", а также создан и назначен этой ссылочной переменной *массив*, имеющий по первому индексу 10 элементов, а по второму 20. То есть мы имеем 10 ячеек типа "*одномерный массив*", каждая из которых ссылается на *массив* из 20 целых чисел. При этом базовым типом для ячеек по первому индексу является `int[]`, а для ячеек по второму индексу `int`.

Рассмотрим работу с двумерными массивами на примере заполнения двумерного массива случайными числами:

```
int m=10;//10 строк
int n=20;//20 столбцов
int[][] a=new int[m][n];
for(int i=0;i<m;i++){ //цикл по строкам
    for(int j=0;j<n;j++){ //цикл по столбцам
        a[i][j]=(int)(100*Math.random());
        System.out.print(a[i][j]+" ");
    };
    System.out.println();//перевод на новую строку после вывода строки матрицы
};
```

Многомерные массивы задаются аналогично двумерным – только указывается необходимое количество прямоугольных скобок. Следует отметить, что массивы размерности больше 3 используют крайне редко.

Обычно в двумерных и многомерных массивах задают одинаковый размер всех массивов, связанных с ячейками по какому-либо индексу. Такие массивы называют регулярными. В *Java*, в отличие от большинства других языков программирования, можно задавать массивы с разным размером массивов, связанных с ячейками по какому-либо индексу. Такие "непрямоугольные" массивы называют иррегулярными. Обычно их используют для экономии памяти. При работе с иррегулярными массивами следует быть особенно аккуратными, так как разный размер "вложенных" массивов часто приводит к ошибкам при реализации алгоритмов.

Пример задания иррегулярного двумерного массива треугольной формы:

```
int n=9;
int[][] a=new int[n][];
for(int i=0;i<a.length;i++){ //цикл по строкам
    a[i]=new int[i+1]; //число элементов в строке равно i+1
    for(int j=0;j<a[i].length;j++){ //цикл по столбцам
        a[i][j]=100*i+j;
```

```
            System.out.print(a[i][j]+" ");
        };
        System.out.println();//перевод на новую строку после вывода строки матрицы
    };
```

После *создания массива* требуется его инициализировать – записать нужные значения в ячейки. До сих пор мы делали это путем задания значений в цикле по некоторой формуле, однако часто требуется задать конкретные значения. Конечно, можно это сделать в виде

```
int[] a=new int[4];
a[0]=2;
a[1]=0;
a[2]=0;
a[3]=6;
```

Но гораздо удобнее следующий вариант синтаксиса:

```
int[] a=new int[] {2,0,0,6};
```

При этом приходится задавать *массив* без указания его размера непосредственно с помощью указания значений в фигурных скобках:

В правой части оператора присваивания стоит так называемый *анонимный массив* – у него нет имени. Такие массивы обычно используют для инициализации, а также при написании кода для различного рода проверок.

Если мы хотим присвоить новые значения, приходится либо присваивать поэлементно, либо создавать новый *объект*:

```
a=new int[] {2,0,0,6};
```

При инициализации двумерных и многомерных массивов используют вложенные массивы, задаваемые с помощью фигурных скобок. Например, фрагмент кода

```
int[][] b= new int[][]
{
  {2,0,0,0}, //это b[0]
  {2,0,0,1}, //это b[1]
  {2,0,0,2}, //это b[2]
  {1,0,0,0}, //это b[3]
  {2,0,0,0}, //это b[4]
  {3,0,0,0}, //это b[5]
};
```

приведет к заданию целочисленного двумерного массива `b`, состоящего из 6 строк и 4 столбцов, т.е. `int[6][4]`. Таким образом можно задавать как регулярные, так и иррегулярные массивы. Но следует помнить, что в таком варианте синтаксиса проверки правильности размера массива по индексам не делается, что может привести к ошибкам. Например, следующий код при компиляции не выдаст ошибки, а будет создан иррегулярный *массив*:

```
int[][] b= new int[][]
{
  {2,0,0,0}, //это b[0]
  {2,0,0,1}, //это b[1]
  {2,0,0,2}, //это b[2]
  {1,0,0,0}, //это b[3]
  {2,0,0,0}, //это b[4]
  {3,0,0}, //это b[5] – массив из трех элементов
};
```

Из объектов-массивов можно вызывать метод *clone()*, позволяющий создавать копию (клон) массива:

```
a=new int[] {2,0,0,6};
int[] a1=a.clone();
```

Напомним, что *присваивание*

```
int[] b=a;
```

не приведет к копированию массива - просто *переменная* `b` станет ссылаться на тот же *объект-массив*. *Копирование* массивов можно осуществлять в цикле, но гораздо быстрее использовать метод `System.arraycopy` .

```
int[] b=new int[a.length+10];
System.arraycopy(a,index1a,b, index1b,count);
```

Из `a` в `b` копируется `count` элементов начиная с индекса `index1a` в массиве `a` . Они размещаются в массиве `b` начиная с индекса `index1b` . Содержимое остальных элементов `b` не меняется. Для использования метода требуется, чтобы *массив* `b` существовал и имел необходимую длину - при выходе за границы массивов возбуждается *исключительная ситуация*.

Быстрое заполнение массива одинаковыми значениями может осуществляться методом `Arrays.fill(массив, значение)` . *Класс* `Arrays` расположен в пакете *java*.util.

Поэлементное *сравнение массива* следует выполнять с помощью метода `Arrays.equals(a,a1)` . Заметим, что у любого массива имеется метод `equals` , унаследованный от класса `Object` и позволяющий сравнивать массивы. Но, к сожалению, метод не переопределен, и сравнение идет по адресам объектов, а не по содержимому. Поэтому `a.equals(a1)` это то же самое, что `a==a1` . Оба сравнения вернут `false` , так как адреса объектов, на которые ссылаются переменные `a` и `a1` , различаются. Напротив, сравнения `a.equals(a3)` и `a==a3` вернут `true` , так как `a` и `a3` ссылаются на один и тот же *объект-массив*.

Сортировка (упорядочение по значениям) массива a производится методами `Arrays.sort(a)` и `Arrays.sort(a,index1,index2)` . Первый из них упорядочивает в порядке возрастания весь *массив*, второй - часть элементов (от индекса `index1` до индекса `index2`). Имеются и более сложные методы сортировки. Элементы массива должны быть сравниваемы (поддерживать операцию сравнения).

`Arrays.deepEquals(a1,a2)` - сравнение на *равенство* содержимого *массивов объектов* `a1` и `a2` путем глубокого сравнения (на *равенство* содержимого, а не ссылок - на произвольном уровне вложенности).

Также в классе `Arrays` содержится большое число других полезных методов.

7.2. Коллекции, списки, итераторы

В *Java* получили широкое использование *коллекции* (Collections) - "умные" массивы с динамически изменяемой длиной, поддерживающие ряд важных дополнительных операций по сравнению с массивами. Базовым для иерархии коллекций является *класс* `java.util.AbstractCollection` . (В общем случае *класс* коллекции не обязан быть потомком `AbstractCollection` - он может является любым классом, реализующим *интерфейс* `Collection`). Основные классы и интерфейсы коллекций:

- интерфейсы `Set, SortedSet` ; классы `HashSet, TreeSet, EnumSet, LinkedHashSet` - множества (неупорядоченные или некоторым образом упорядоченные наборы неповторяющихся элементов);
- интерфейс `java.util.List` ; классы `java.awt.List, ArrayList, LinkedList, Vector` - списки (упорядоченные наборы элементов, причем элементы могут повторяться в разных местах списка);
- интерфейсы `Map, SortedMap` ; классы `HashMap, HashTable, ConcurrentHashMap, TreeMap, EnumMap, Properties` - таблицы (списки пар "имя-значение").

Доступ к элементам коллекции в общем случае не может осуществляться по индексу, так как не все коллекции поддерживают индексацию элементов. Эту функцию осуществляют с помощью специального объекта - *итератора* (`iterator`). У каждой коллекции `collection` имеется свой *итератор* который умеет с ней работать, поэтому *итератор* вводят следующим образом:

```
Iterator iter = collection.iterator()
```

У итераторов имеются следующие три метода:

- `boolean hasNext()` - дает информацию, имеется ли в коллекции следующий объект.
- `Object next()` - возвращает ссылку на следующий объект коллекции.
- `void remove()` - удаляет из коллекции текущий объект, то есть тот, ссылка на который была получена последним вызовом `next()` .

Пример преобразования массива в коллекцию и цикл с доступом к элементам этой коллекции, осуществляемый с помощью *итератора*:

```
java.util.List components= java.util.Arrays.asList(this.getComponents());

for (Iterator iter = components.iterator();iter.hasNext();) {
      Object elem = (Object) iter.next();
      javax.swing.JOptionPane.showMessageDialog(null,"Компонент: "+
                                                elem.toString());
}
```

Таблица 7.1. Основные методы коллекций:

Имя метода	Действие
`boolean add(Object obj)`	Добавление объекта в коллекцию (в конец списка). Возвращает `true` в случае успешного добавления – изменения коллекции. Коллекция может не позволить добавление элементов несовместимого типа или не подходящих по какому-либо другому признаку.
`boolean addAll(Collection c)`	Добавление в коллекцию всех объектов из другой коллекции. Возвращает `true` в случае успешного добавления, то есть если добавлен хотя бы один элемент.
`void clear()`	Очистка коллекции – удаление из нее ссылок на все входящие в коллекцию объекты. При этом те объекты, на которые имеются ссылки у других элементов программы, не удаляются из памяти.
`boolean contains(Object obj)`	Возвращает `true` в случае, если коллекция содержит объект `obj`. Проверка осуществляется с помощью поочередного вызова метода `obj.equals(e)` для элементов `e`, входящих в коллекцию.
`boolean containsAll(Collection c)`	Возвращает `true` в случае, если коллекция содержит все элементы коллекции `c`.
`boolean isEmpty()`	Возвращает `true` в случае, если коллекция пуста, то есть не содержит ни одного элемента.
`Iterator iterator()`	Возвращает ссылку на *итератор* – объект, позволяющий получать поочередный доступ к элементам коллекции. Для одной коллекции разрешается иметь произвольное число объектов-итераторов, в том числе – разных типов. В процессе работы они могут указывать на разные элементы коллекции. После создания *итератор* всегда указывает на начало коллекции – вызов его метода `next()` дает ссылку на начальный элемент коллекции.
`boolean remove(Object obj)`	Удаляет из коллекции первое встретившееся вхождение объекта `obj`. Поиск и удаление осуществляется с помощью *итератора*. Возвращает `true` в случае, если удаление удалось, то есть если коллекция изменилась.
`boolean removeAll(Collection c)`	Удаляет из коллекции все элементы коллекции `c`. Возвращает `true` в случае, если удаление удалось, то есть если коллекция изменилась.
`boolean`	Оставляет в коллекции только те из входящих в нее

`retainAll(Collection c)`	элементов, которые входят в коллекцию `c`.
`int size()`	Возвращает число элементов в коллекции.
`Object[] toArray()`	Возвращает массив ссылок на объекты, содержащиеся в коллекции. То есть преобразует коллекцию в массив.
`T[] toArray(T[n] a)`	Возвращает массив элементов типа `T`, полученных в результате преобразования элементов, содержащихся в коллекции. То есть преобразует коллекцию в массив. Если число элементов коллекции не превышает размер `n` массива `a`, размещение данных производится в существующих ячейках памяти, отведенных под массив. Если превышает `n` – в памяти динамически создается и заполняется новый набор ячеек, и их число делается равным числу элементов коллекции. После чего переменная `a` начинает ссылаться на новый набор ячеек.
`String toString()`	Метод переопределен – он возвращает строку со списком элементов коллекции. В списке выводятся заключенные в квадратные скобки строковые представления элементов, разделяемые комбинацией "," – запятая с пробелом после нее.

Самыми распространенными вариантами коллекций являются списки (`Lists`). Они во многом похожи на массивы, но отличаются от массивов тем, что в списках основными операциями являются добавление и удаление элементов. А не *доступ* к элементам по индексу, как в массивах.

В классе `List` имеются методы коллекции, а также ряд дополнительных методов:

- `list.get(i)` – получение ссылки на элемент списка `list` по индексу `i`.
- `list.indexOf(obj)` – получение индекса элемента `obj` в списке `list`. Возвращает -1 если объект не найден.
- `list.listIterator()` – получение ссылки на *итератор* типа `ListIterator`, обладающего дополнительными методами по сравнению с *итераторами* типа `Iterator`.
- `list.listIterator(i)` – то же с позиционированием *итератора* на элемент с индексом `i`.
- `list.remove(i)` – удаление из списка элемента с индексом `i`.
- `list.set(i,obj)` – замена в списке элемента с индексом `i` на объект `obj`.
- `list.subList(i1,i2)` – возвращает ссылку на подсписок, состоящий из элементов списка с индексами от `i1` до `i2`.

Кроме них в классе `List` имеются и многие другие полезные методы.

Ряд полезных методов для работы с коллекциями содержится в классе `Collections`:

- `Collections.addAll(c,e1,e2,…,eN)` – добавление в коллекцию с произвольного числа элементов `e1,e2,…,eN`.
- `Collections.frequency(c,obj)` – возвращает число вхождений элемента `obj` в коллекцию `c`.
- `Collections.reverse(list)` – обращает порядок следования элементов в списке `list` (первые становятся последними и наоборот).
- `Collections.sort(list)` – сортирует список в порядке возрастания элементов. Сравнение идет вызовом метода `e1.compareTo(e2)` для очередных элементов списка `e1` и `e2`.

Кроме них в классе `Collections` имеются и многие другие полезные методы.

В классе `Arrays` имеется метод

- `Arrays.asList(a)` – возвращает ссылку на список элементов типа `T`, являющийся оболочкой над массивом `T[] a`. При этом и массив, и список содержат одни и те же элементы, и изменение элемента списка приводит к изменению элемента массива, и наоборот.

7.3. Работа со строками в Java. Строки как объекты. Классы String, StringBuffer и StringBuilder

Класс `String` инкапсулирует действия со строками. *Объект* типа `String` – строка, состоящая из произвольного числа символов, от 0 до $2 * 10^9$. Литерные *константы* типа `String` представляют собой последовательности символов, заключенные в двойные кавычки: "A", "*abcd*", "*abcd*", "Мама моет раму", " ".

Это так называемые "длинные" строки. Внутри литерной строковой *константы* не разрешается использовать ряд символов – вместо них применяются *управляющие* последовательности.

Внутри строки разрешается использовать переносы на новую строку. Но литерные *константы* с такими переносами запрещены, и надо ставить *управляющую последовательность* `"\n"`. К сожалению, такой перенос строки не срабатывает в компонентах.

Разрешены пустые строки, не содержащие ни одного символа.

В языке *Java строковый* и *символьный тип* несовместимы. Поэтому `"A"` – строка из одного символа, а `'A'` – число с *ASCII* кодом символа `"A"`. Это заметно усложняет работу со строками и символами.

Строки можно складывать: если `s1` и `s2` строковые литерные *константы* или переменные, то результатом *операции* `s1+s2` будет строка, являющаяся сцеплением (конкатенацией) строк, хранящихся в `s1` и `s2`. Например, в результате *операции*

```
String s="Это "+"моя строка";
```

в переменной s будет храниться строковое *значение* "Это моя строка".

Для строк разрешен оператор `"+="`. Для строковых операндов `s1` и `s2` *выражение* `s1+=s2` *эквивалентно выражению* `s1=s1+s2`.

Любая строка (набор символов) является объектом – экземпляром класса `String`. Переменные типа `String` являются ссылками на объекты, что следует учитывать при *передаче параметров строкового типа* в подпрограммы, а также при многократных изменениях строк. При каждом изменении строки в динамической области памяти создается новый *объект*, а прежний превращается в "мусор". Поэтому при многократных изменениях строк в цикле возникает много мусора, что нежелательно.

Очень частой ошибкой является попытка сравнения строк с помощью оператора `"=="`. Например, результатом выполнения следующего фрагмента

```
String s1="Строка типа String";
String s2="Строка";
s2+=" типа String";
if(s1==s2)
    System.out.println("s1 равно s2");
else
    System.out.println("s1 не равно s2");
```

будет *вывод* в консольное окно строки `"s1 не равно s2"`, так как объекты-строки имеют в памяти разные адреса. Сравнение по содержанию для строк выполняет оператор `equals`. Поэтому если бы вместо `s1==s2` мы написали `s1.equals(s2)`, то получили бы ответ `"s1 равно s2"`.

Удивительным может показаться факт, что результатом выполнения следующего фрагмента

```
String s1="Строка";
String s2="Строка";
if(s1==s2)
    System.out.println("s1 равно s2");
else
    System.out.println("s1 не равно s2");
```

будет *вывод* в консольное окно строки "s1 равно s2". Дело в том, что оптимизирующий *компилятор Java* анализирует имеющиеся в коде программы литерные *константы*, и для одинаковых по содержанию констант использует одни и те же объекты-строки.

В остальных случаях то, что строковые переменные ссылочные, обычно никак не влияет на работу со строковыми переменными, и с ними можно действовать так, как если бы они содержали сами строки.

В классе `String` имеется ряд методов. Перечислим важнейшие из них. Пусть `s1` и `subS` имеют тип `String`, `charArray` – *массив символов* `char[]`, `ch1` – *переменная* или *значение* типа `char`, а `i`, `index1` и `count` ("счет, количество") – целочисленные переменные или значения. Тогда

`String.valueOf(параметр)` – возвращает строку типа `String`, являющуюся результатом преобразования параметра в строку. Параметр может быть любого примитивного или *объектного типа*.

`String.valueOf(charArray, index1,count)` – функция, аналогичная предыдущей для массива символов, но преобразуется `count` символов начиная с символа, имеющего индекс `index1`.

У объектов типа `String` также имеется ряд методов. Перечислим важнейшие из них.

`s1.charAt(i)` – символ в строке `s1`, имеющий индекс `i` (индексация начинается с нуля).

`s1.endsWith(subS)` – возвращает `true` в случае, когда строка `s1` заканчивается последовательностью символов, содержащихся в строке `subS`.

`s1.equals(subS)` – возвращает `true` в случае, когда последовательностью символов, содержащихся в строке `s1`, совпадает с последовательностью символов, содержащихся в строке `subS`.

`s1.equalsIgnoreCase(subS)` – то же, но при сравнении строк игнорируются различия в регистре символов (строчные и заглавные буквы не различаются).

`s1.getBytes()` – возвращает массив типа `byte[]`, полученный в результате платформо-зависимого преобразования символов строки в последовательность байт.

`s1.getBytes(charset)` – то же, но с указанием кодировки (`charset`). В качестве строки `charset` могут быть использованы значения "*ISO-8859*-1" (стандартный латинский алфавит в 8-битовой кодировке), "UTF-8", "*UTF-16*" (символы UNICODE) и другие.

`s1.indexOf(subS)` – индекс позиции, где в строке `s1` первый раз встретилась последовательность символов `subS`.

`s1.indexOf(subS,i)` – индекс позиции начиная с `i`, где в строке `s1` первый раз встретилась последовательность символов `subS`.

`s1. lastIndexOf (subS)` – индекс позиции, где в строке `s1` последний раз встретилась последовательность символов `subS`.

`s1. lastIndexOf (subS,i)` – индекс позиции начиная с `i`, где в строке `s1` последний раз встретилась последовательность символов `subS`.

`s1.length()` – длина строки (число 16-битных символов UNICODE, содержащихся в строке). Длина пустой строки равна нулю.

`s1.replaceFirst(oldSubS,newSubS)` – возвращает строку на основе строки `s1`, в которой произведена замена первого вхождения символов строки `oldSubS` на символы строки `newSubS`.

`s1.replaceAll(oldSubS,newSubS)` – возвращает строку на основе строки `s1`, в которой произведена замена всех вхождений символов строки `oldSubS` на символы строки `newSubS`.

`s1.split(separator)` – возвращает массив строк `String[]`, полученный разделением строки `s1` на независимые строки по местам вхождения сепаратора, задаваемого строкой `separator`. При этом символы, содержащиеся в строке `separator`, в получившиеся строки не входят. Пустые строки из конца получившегося массива удаляются.

`s1.split(separator, i)` – то же, но положительное `i` задает максимальное допустимое число элементов массива. В этом случае последним элементом массива становится окончание строки `s1`, которое не было расщеплено на строки, вместе с входящими в это окончание символами сепараторов. При `i` равном 0 ограничений нет, но пустые строки из конца получившегося массива удаляются. При `i <0` ограничений нет, а пустые строки из конца получившегося массива не удаляются.

`s1.startsWith(subS)` – возвращает `true` в случае, когда строка `s1` начинается с символов строки `subs`.

`s1.startsWith(subs, index1)` – возвращает `true` в случае, когда символы строки `s1` с позиции `index1` начинаются с символов строки `subs`.

`s1.substring(index1)` – возвращает строку с символами, скопированными из строки `s1` начиная с позиции `index1`.

`s1.substring(index1,index2)` – возвращает строку с символами, скопированными из строки `s1` начиная с позиции `index1` и кончая позицией `index2`.

`s1.toCharArray()` – возвращает *массив символов*, скопированных из строки `s1`.

`s1.toLowerCase()` – возвращает строку с символами, скопированными из строки `s1`, и преобразованными к нижнему регистру (строчным буквам). Имеется вариант метода, делающего такое преобразование с учетом конкретной кодировки (`locale`).

`s1.toUpperCase()` – возвращает строку с символами, скопированными из строки `s1`, и преобразованными к верхнему регистру (заглавным буквам). Имеется вариант метода, делающего такое преобразование с учетом конкретной кодировки (`locale`).

`s1.trim()` – возвращает копию строки `s1`, из которой убраны ведущие и завершающие пробелы.

В классе `Object` имеется метод `toString()`, обеспечивающий строковое *представление* объекта. Конечно, оно не может отражать все особенности объекта, а является представлением "по мере возможностей". В самом классе `Object` оно обеспечивает возврат методом полного имени класса (квалифицированное именем пакета), затем идет символ "@", после которого следует число – *хэш-код* объекта (число, однозначно характеризующее данный *объект* во время сеанса работы) в шестнадцатеричном представлении. Поэтому во всех классах-наследниках, где этот метод не переопределен, он возвращает такую же конструкцию. Во многих стандартных классах этот метод переопределен. Например, для числовых классов метод `toString()` обеспечивает *вывод* строкового представления соответствующего числового значения. Для строковых объектов – возвращает саму строку, а для *символьных (тип* `Char`) – символ.

При использовании операций `"+"` и `"+="` с операндами, один из которых является строковым, а другой нет, метод `toString()` вызывается автоматически для нестрокового операнда. В результате получается *сложение* (*конкатенация*) двух строк. При таких действиях следует быть очень внимательными, так как результат сложения более чем двух слагаемых может оказаться сильно отличающимся от ожидаемого. Например,

```
String s=1+2+3;
```

даст вполне *ожидаемое значение* `s=="6"`. А вот *присваивание*

```
String s="Сумма ="+1+2+3;
```

даст не очень понятное начинающим программистам *значение* " Сумма =123". Дело в том, что в первом случае сначала выполняются арифметические сложения, а затем результат преобразуется в строку и присваивается левой части. А во втором сначала производится *сложение* "Сумма ="+1. Первый *операнд строковый*, а второй – числовой. Поэтому для второго операнда вызывается метод `toString()`, и складываются две строки. Результатом будет строка "Сумма =1". Затем складывается строка "Сумма =1" и число 2. Опять для второго операнда вызывается метод `toString()`, и складываются две строки. Результатом будет строка "Сумма =12". Совершенно так же выполняется *сложение строки* "Сумма =12" и числа 3.

Еще более странный результат получится при присваивании

```
String s=1+2+" не равно "+1+2;
```

Следуя изложенной выше логике мы получаем, что результатом будет строка "3 не равно 12".

Выше были приведены простейшие примеры, и для них все достаточно очевидно. Если же в такого рода выражениях используются числовые и *строковые функции*, да еще с оператором `" ? : "`, результат может оказаться совершенно непредсказуемым.

Кроме указанных выше имеется ряд строковых операторов, заданных в оболочечных числовых классах. Например, мы уже хорошо знаем методы преобразования строковых представлений чисел в числовые значения

`Byte.parseByte` (*строка*)
`Short.parseShort` (*строка*)
`Integer.parseInt` (*строка*)
`Long.parseLong` (*строка*)
`Float.parseFloat` (*строка*)
`Double.parseDouble` (*строка*)

и метод `valueOf` (*строка*), преобразующий строковые представления чисел в числовые объекты – экземпляры оболочечных классов `Byte, Short, Character, Integer, Long, Float, Double`. Например,

`Byte.valueOf` (*строка*), и т.п.

Кроме того, имеются методы классов `Integer` и `Long` для преобразования чисел в двоичное и шестнадцатеричное строковое *представление*:

`Integer.toBinaryString` (*число*)
`Integer.toHexString` (*число*)
`Long.toBinaryString` (*число*)
`Long.toHexString` (*число*)

Имеется возможность *обратного преобразования* – из строки в *объект* соответствующего класса (`Byte, Short, Integer, Long`) с помощью метода `decode`:

`Byte.decode` (*строка*), и т.п.

Также полезны методы для анализа отдельных символов:

`Character.isDigit` (*символ*) – булевская функция, проверяющая, является ли символ цифрой.

`Character.isLetter` (*символ*) – булевская функция, проверяющая, является ли символ буквой.

`Character.isLetterOrDigit` (*символ*) – булевская функция, проверяющая, является ли символ буквой или цифрой.

`Character.isLowerCase` (*символ*) – булевская функция, проверяющая, является ли символ символом в нижнем регистре.

`Character.isUpperCase` (*символ*) – булевская функция, проверяющая, является ли символ символом в верхнем регистре.

`Character.isWhitespace` (*символ*) – булевская функция, проверяющая, является ли символ "пробелом в широком смысле" – пробелом, символом табуляции, перехода на новую строку и т.д.

Для того чтобы сделать работу с многочисленными присваиваниями более эффективной, используются классы `StringBuffer` и `StringBuilder`. Они особенно удобны в тех случаях, когда требуется проводить изменения внутри одной и той же строки (убирать или вставлять символы, менять их местами, заменять одни на другие). Изменение значений переменных этого класса не приводит к созданию мусора, но несколько медленнее, чем при работе с переменными типа `String`. *Класс* `StringBuffer` рекомендуется использовать в тех случаях, когда используются потоки (`threads`) – он, в отличие от классов `String` и `StringBuilder`, обеспечивает синхронизацию строк. *Класс* `StringBuilder`, введенный начиная с *JDK* 1.5, полностью ему подобен, но синхронизации не поддерживает. Зато обеспечивает большую скорость работы со строками (что обычно бывает важно только в лексических анализаторах).

К сожалению, совместимости по присваиванию между переменными этих классов нет, как нет и возможности преобразования этих типов. Но в классах `StringBuffer` и `StringBuilder` имеется метод `sb.append(s)`, позволяющий добавлять в конец "буферизуемой" строки `sb` обычную строку `s`. Также имеется метод `sb.insert(index,s)`, позволяющий вставлять начиная с места символа, имеющего *индекс* `index`, строку `s`.

Пример:

```
StringBuffer sb=new StringBuffer();
sb.append("типа StringBuffer");
sb.insert(0,"Строка ");
System.out.println(sb);
```

Буферизуемые и обычные строки можно сравнивать на совпадение содержания: `s1.contentEquals(sb)` – булевская *функция*, возвращающая `true` в случае, когда строка `s1` содержит такую же последовательность символов, как и строка `sb`.

7.4. Работа с графикой

Вывод графики осуществляется с помощью объектов типа `java.awt.Graphics`. Для них определен ряд методов, описанных в следующей далее таблице.

Подразумевается, что `w` – ширина области или фигуры, `h` – *высота*; `x,y` – *координаты* левого верхнего угла области. Для фигуры `x,y` – *координаты* левого верхнего угла прямоугольника, в который вписана фигура.

Таблица 7.2.

Параметры вывода графики

`Color getColor()`	Узнать текущий цвет рисования.
`setColor(Color c)`	Задать текущий цвет рисования.
`Font getFont()`	Узнать текущий шрифт для вывода текстовой информации.
`setFont(Font f)`	Установить текущий шрифт для вывода текстовой информации. Экземпляр шрифта создается с помощью конструктора Font("имяШрифта",стильШрифта,размерШрифта)
`FontMetrics getFontMetrics()`	Узнать параметры текущего шрифта
`FontMetrics getFontMetrics(Font f)`	Узнать параметры для произвольного шрифта `f`

`setXORMode(Color c1)`	Установка режима рисования `XOR` ("исключающее или") для цвета `c1`. При этом вывод точки цвета `color` дает цвет, равный побитовому значению `color ^ c1` (то есть color XOR c1) для числовой RGB-кодировки цвета. Повторный вывод графического изображения на то же место приводит к восстановлению первоначального изображения в области вывода.
`setPaintMode()`	Возврат в обычный режим из режима рисования XOR.
`translate(x0,y0)`	Сдвиг начала координат графического контекста в точку `x0,y0`. Все координаты, указанные при выводе графических примитивов, отсчитываются относительно этого начала координат.

Рисование контурных фигур

`drawLine(x1,y1,x2,y2)`	Вывод линии из точки с координатами `x1,y1` в точку `x2,y2`
`drawRect(x,y,w,h)`	Вывод прямоугольника.
`drawRoundRect(x,y,w,h,arcWidth,arcHeight)`	Вывод скругленного прямоугольника.
`draw3DRect(x,y,w,h,isRaised)`	Вывод "объемного" прямоугольника. Если переменная `isRaised ==true`, он "выпуклый" (`raised`), иначе - "вдавленный".
`drawPolygon(Polygon p)` ; `drawPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Вывод многоугольника по массиву точек, `nPoints` - число точек.
`drawPolyline(int[] xPoints,int[] yPoints, int nPoints)`	Вывод незамкнутой ломаной линии по массиву точек, `nPoints` - число точек.
`drawOval(x,y,w,h)`	Вывод эллипса.
`drawArc(x,y,w,h,startAngle,arcAngle)`	Вывод дуги эллипса. Начальный угол `startAngle` и угол, задающий угловой размер дуги `arcAngle`, задаются в градусах.
`drawImage(Image img,int x,int y, ImageObserver observer)` и другие перегруженные варианты метода	Вывод изображения.

Рисование заполненных фигур

`clearRect(x,y,w,h)`	Очистка прямоугольника (заполнение текущим цветом)
`fillRect(x,y,w,h)`	Вывод прямоугольника, заполненного текущим цветом.
`fillRoundRect(x,y,w,h,arcWidth,arcHeight)`	Вывод скругленного прямоугольника, заполненного текущим цветом.
`fill3DRect(x,y,w,h, isRaised)`	Вывод "объемного" прямоугольника, заполненного текущим цветом. Если

	переменная `isRaised ==true` , он "выпуклый" (raised), иначе - "вдавленный".
`fillPolygon(Polygon p)` `fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Вывод многоугольника, заполненного текущим цветом.
`fillOval(x,y,w,h)`	Вывод эллипса, заполненного текущим цветом.
`fillArc(x,y,w,h,startAngle,arcAngle)`	Вывод сектора эллипса, заполненной текущим цветом. Заполняется сектор, ограниченный отрезками из центра эллипса в концы дуги, и самой дугой.
`copyArea(x,y,w,h,dx,dy)`	Копирование области на новое место, сдвинутое от старого на `dx,dy`
Вывод текстовой информации	
`drawString(s,x,y)`	Вывод строки s
`drawChars(char[] data,int offset,int length,int x,int y)`	Вывод массива символов
`drawBytes(byte[] data,int offset,int length,int x,int y)`	Вывод символов, представленных как последовательность байт
Управление областью вывода	
`setClip(x,y,w,h)` `setClip(Shape clip)`	Установка новых границ области вывода. Вне этой области при выводе графических примитивов они усекаются (не выводятся).
`clipRect(x,y,w,h)`	Сужение области вывода.
`Rectangle getClipBounds()` `Rectangle getClipBounds(Rectangle r)`	Возвращает параметры прямоугольника, в который вписана область вывода.
`Graphics create()`	`g1=g.create()` - создание копии графического объекта `g`
`dispose()`	*Деструктор* - уничтожение графического объекта с одновременным высвобождением ресурсов (без ожидания, когда это сделает сборщик мусора).

Пример метода, работающего с графикой.

```
java.awt.Graphics g,g1;

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
  java.awt.Graphics g,g1;
  g=jPanel1.getGraphics();
  int x1=20,x2=120,y1=20,y2=120;
  int x3=20,y3=20,w3=60,h3=80;
  int x4=30,y4=60,w4=30,h4=40;
  int x0=10,y0=10,w0=10,h0=10;
  int w1=80,h1=120;
  g.setClip(0,0,60,80);//границы области вывода
  g.drawLine(x1,y1,x2,y2);//линия
  g.drawOval(x3,y3,w3,h3);//эллипс
  g.clipRect(x4,y4,20,20);//сужение области вывода
```

```
    g.clearRect(x4,y4,w4,h4);//очистка прямоугольника
    g.setClip(0,0,200,280); //новые границы области вывода
    g.copyArea(x1,y1,w1,h1,60,0);
    g.draw3DRect(10,20,w1,h1,false);
    g.drawPolygon(new java.awt.Polygon(new int[]{10,10,20,40},
                                       new int[]{10,20,30,60},4) );
  }
```

В случае попытки такого использования возникает проблема: при перерисовке графического контекста все выведенное изображение исчезает. А перерисовка вызывается автоматически при изменении размера окна приложения, а также его восстановлении после минимизации или перекрытия другим окном.

Для того, чтобы результаты вывода не пропадали, в *классе приложения* требуется переопределить метод `paint` , вызываемый при отрисовке. Код этого метода может выглядеть так:

```
public void paint(java.awt.Graphics g){
   super.paint(g);
   g=jPanel1.getGraphics();
   ... - команды графического вывода
}
```

Правда, при изменении размера окна приложения этот код не сработает, и для панели надо будет назначить обработчик

```
private void jPanel1ComponentResized (java.awt.event.ComponentEvent evt) {
 ... - команды графического вывода
}
```

То, что для изменения размера компонента следует писать отдельный обработчик, вполне разумно - ведь при восстановлении окна требуется только воссоздать изображение прежнего размера. А при изменении размера может потребоваться масштабирование выводимых элементов. Поэтому алгоритмы вывода графики в этих случаях заметно отличаются.

В случае отрисовки из обработчика какого-либо события изменения графического контекста не происходит до окончания обработчика. Это принципиальная особенность работы по идеологии обработчиков событий - пока не кончится один обработчик, следующий не начинается. Для досрочной отрисовки непосредственно во *время выполнения* обработчика события служит *вызов метода* `update(Graphics g)` . Пример:

```
for(int i=0;i<=100;i++){
  FiguresUtil.moveFigureBy(figure,dx,dy);
  update(g);
};
```

При работе со статическими изображениями изложенных алгоритмов вполне достаточно. Однако при использовании движущихся элементов во многих *графических системах* возникает мельтешение, связанное с постоянными перерисовками. В этих случаях обычно применяют идеологию двойной буферизации: отрисовку элементов по невидимому буферному изображению, а затем показ этого изображения в качестве видимого. А то изображение, которое было видимо, при этом становится невидимым буфером.

7.5. Исключительные ситуации

Обработка исключительных ситуаций

При работе программы выполнение операторов обычно идет в рамках "основного ствола" - в случае, когда все идет как надо. Но время от времени возникают *исключительные ситуации* (*исключения* - exceptions), приводящие к ответвлению от основного ствола: деление на 0, отсутствие места на диске или попытка писать на защищенную для записи дискету, ввод с клавиатуры ошибочного символа (например, буквы вместо цифры). В отличие от катастрофических ситуаций (ошибок) такие ситуации в большинстве случаев могут быть учтены в программе, и, в частности, они не должны приводить к аварийному завершению программы.

В языках программирования предыдущих поколений для решения указанных проблем приходилось использовать огромное число проверок на допустимость присваиваний и математических операций. Мало того, что эти проверки резко замедляли работу программы - не было гарантии, что они достаточны, и что во время работы программы не возникнет "вылет" из-за возникновения непредусмотренной ситуации.

В Java, как и в других современных языках программирования, для таких целей предусмотрено специальное средство – обработка исключительных ситуаций. При этом используется так называемый защищенный блок программного кода `try` ("попытаться"), после которого следует необязательные блоки перехвата исключений `catch`, за которыми идет необязательный блок очистки ресурсов `finally`.

Про наступившую исключительную ситуацию говорят, что она возникает, либо – что она возбуждается. В английском языке для этого используется слово `throw` – "бросить". Поэтому иногда в переводной литературе используют дословный перевод "бросается исключительная ситуация".

Общий случай использования защищенного блока программного кода и перехвата исключительных ситуаций выглядит так:

```
try{
 операторы0;
}
catch (ТипИсключения1 переменная1){
 операторы1;
}
catch (ТипИсключения2 переменная2){
 операторы2;
}
catch (ТипИсключенияN переменнаяN){
 операторыN;
}
finally{
 операторы;
}
```

Отметим, что при задании блоков `try-catch-finally` после фигурных скобок точку с запятой ";" можно не ставить, как и всегда в случае использования фигурных скобок. Но можно и ставить – по усмотрению программиста.

Если исключительных ситуаций не было, `операторы0` в блоке `try` выполняются в обычном порядке, после чего выполняются `операторы` в блоке `finally`. Если же возникла исключительная ситуация в блоке `try`, выполнение блока прерывается, и идет перехват исключений в блоках `catch` ("перехватить"). В качестве параметра оператора `catch` задается ссылочная переменная, имеющая тип той исключительной ситуации, которую должен *перехватить данный* блок. Чаще всего эту переменную называют `e` (по первой букве от exception). Если *тип исключения* совместим с типом, указанном в качестве параметра, выполняется соответствующий оператор. После чего проверок в следующих блоках `catch` не делается.

После проверок и, возможно, перехвата исключения в блоках `catch` выполняются операторы блока `finally`. Его обычно используют для высвобождения ресурсов, и поэтому часто называют блоком "очистки ресурсов". Специальных операторов или зарезервированных конструкций для обработки в блоке `finally` нет. Отличие кода внутри блока `finally` от кода, стоящего после оператора `try…finally`, возникает только при наличии внутри блоков `try` или `catch` операторов `break`, `continue`, `return` или `System.exit`, то есть операторов, прерывающих работу блока программного кода. В этом случае независимо от их срабатывания или несрабатывания сначала происходит выполнение операторов блока `finally`, и только потом происходит переход в другое место программы в соответствии с оператором прерывания.

Пример обработки исключений:

```
void myETest(String s,double y){
    double x, z;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(x/y);
    } catch(ArithmeticException e){
        System.out.println("Деление на ноль");
    } catch(NumberFormatException e){
        System.out.println("Корень из отрицательного числа!");
```

```
    }
};
```

Иерархия исключительных ситуаций

Исключительные ситуации в Java являются объектами. Их типы являются классами-потомками *объектного типа* `Throwable` (от throw able – "способный возбудить исключительную ситуацию"). От `Throwable` наследуются классы `Error` ("Ошибка") и `Exception` ("Исключение"). Экземплярами класса `Error` являются непроверяемые исключительные ситуации, которые невозможно перехватить в блоках `catch`. Такие исключительные ситуации представляют катастрофические ошибки, после которых невозможна нормальная работа приложения. Экземплярами класса `Exception` и его потомков являются проверяемые исключительные ситуации. Кроме одного потомка – класса `RuntimeException` (и его потомков). Имя этого класса переводится как "Исключительные ситуации времени выполнения".

Классы исключительных ситуаций либо предопределены в стандартных пакетах (существуют исключительные ситуации `ArithmeticException` для арифметических операций в пакете java.lang, IOException в пакете java.io, и так далее), либо описываются пользователем как потомки класса `Exception` или его потомков.

В Java типы-исключения принято именовать, оканчивая имя класса на `"Exception"` ("Исключение") для *проверяемых исключений* или на `"Error"` ("Ошибка") для непроверяемых.

По правилу *совместимости типов* исключительная ситуация типа-потомка всегда может быть обработана как исключение прародительского типа. Поэтому порядок следования блоков `catch` имеет большое значение: обработчик исключения *более общего* типа следует писать всегда *после* обработчика для его типа-потомка, иначе обработчик потомка никогда не будет вызван.

В приведенном выше примере вместо `NumberFormatException` можно поставить `Exception`, так как других типов исключений кроме `NumberFormatException` сюда доходить не может. В этом случае метод выглядит так:

```
void myETest(String s,double y){
    double x, z;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(x/y);
    } catch(ArithmeticException e){
        System.out.println("Деление на ноль");
    } catch(Exception e){
        System.out.println("Корень из отрицательного числа!");
    }
};
```

Но если бы мы попробовали таким же образом заменить *тип исключения* в первом блоке `catch`, то блок для исключений типа `Exception` всегда перехватывал бы управление, и обработчик для `NumberFormatException` никогда бы не срабатывал. Пример такого неправильно написанного кода:

```
void myETest(String s,double y){
    double x, z;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(x/y);
    } catch(Exception e){
        System.out.println("Деление на ноль");
    } catch(NumberFormatException e){
        System.out.println("Корень из отрицательного числа!");
    }
};
```

В таких случаях среда разработки `NetBeans` выдаст сообщение вида "exception java.lang.NumberFormatException has already been caught" – "исключение java.lang.NumberFormatException уже было перехвачено".

Объявление типа исключительной ситуации и оператор throw

Для того чтобы задать собственный тип исключительной ситуации, требуется задать соответствующий класс. Он должен быть наследником от какого-либо класса исключительной ситуации.

Например, зададим *класс исключения*, возникающего при неправильном вводе пользователем пароля:

```
class WrongPasswordException extends Exception {
   WrongPasswordException(){ // конструктор
     System.out.println("Wrong password!");
   }
}
```

Создание объекта-исключения может проводиться в произвольном месте программы обычным образом, как для всех объектов, при этом возбуждения исключения не происходит.

Программное возбуждение исключительной ситуации производится с помощью оператора `throw`, после которого указывается оператор создания объекта-исключения:

```
throw new ТипИсключения();
```

Например,

```
throw new WrongPasswordException();
```

Если после частичной обработки требуется повторно возбудить исключительную ситуацию `e`, используется вызов

```
throw e;
```

Для *проверяемых исключений* всегда требуется явное возбуждение. При возбуждении исключения во время выполнения какого-либо метода прерывается основной ход программы, и идет процесс обработки исключения. Его иногда называют "всплыванием" исключения по аналогии со всплыванием пузырька. Если в методе исключение данного типа не перехватывается, исполняется соответствующий блок `finally`, если он есть, и всплывание продолжается – происходит выход из текущего метода в метод более высокого уровня. Соответственно, исключение начинает обрабатываться на уровне этого метода. Если оно не перехватывается, происходит выход на еще более высокий уровень, и так далее.

Если в методе исключение данного типа перехватывается, всплывание прекращается. Если же ни на каком уровне исключение не перехвачено – оно обрабатывается исполняющей средой. При этом выполняются действия, предусмотренные в конструкторе исключения, а затем система выводит служебную информацию о том, где и какого типа исключение возникло.

Исключительная ситуация – это особая, экстремальная ситуация, не планируемая заранее. *Не следует использовать исключения в качестве конструкций, на которых основаны часто повторяющиеся в программе действия*.

Задание: усовершенствовать класс `WrongPasswordException` таким образом, чтобы сообщение об ошибке появлялось в виде диалогового окна.

Объявление метода, который может возбуждать исключительную ситуацию. Зарезервированное слово throws

Формат *объявления функции*, которая может возбуждать проверяемые исключительные ситуации, следующий:

```
Модификаторы Тип Имя(список параметров)
 throws ТипИсключения1, ТипИсключения2,…, ТипИсключенияN
{
  Тело функции
}
```

Аналогичным образом объявляется конструктор, который может возбуждать проверяемые исключительные ситуации:

```
Модификаторы ИмяКласса(список параметров)
 throws ТипИсключения1, ТипИсключения2,…, ТипИсключенияN
{
```

```
    Тело конструктора
  }
```

Слово **throws** означает "возбуждает исключительную ситуацию" (дословно – "бросает").

Непроверяемые исключения генерируются и обрабатываются системой автоматически – как правило, приводя к завершению приложения. При этом их типы нигде не указываются, и слово `throws` в заголовке метода указывать не надо.

Если в теле реализуемого метода используется вызов метода, который может возбуждать исключительную ситуацию, и это исключение не перехватывается, в заголовке реализуемого метода требуется указывать соответствующий тип возбуждаемого исключения. Если же это исключение порождается внутри защищенного блока программного кода, и в каком-либо блоке `catch` перехватывается этот *тип исключения* или более общий (прародительский), то указывать в заголовке *тип исключения* не следует.

Как мы уже знаем, в *теле метода* может быть использован оператор `throw`, возбуждающий исключительную ситуацию. В этом случае если объект-исключение не перехвачен, в заголовке метода требуется указывать соответствующий тип возбуждаемого исключения. В частности, оператор `throw` может быть использован для порождения исключения другого типа после обработки перехваченного исключения в блоке `catch`.

Если в *родительском классе* задан метод, в заголовке которого указан тип какой-либо исключительной ситуации, а в классе-потомке этот метод переопределяется, в переопределяемом методе также требуется указывать *совместимый тип* исключительной ситуации. При этом может быть указан *либо тот же тип*, *либо тип* исключительной ситуации – *потомка* от данного типа. В противном случае на этапе компиляции выдается диагностика ошибки.

Пример: проверка пароля, введенного пользователем.

```
class CheckPasswordDemo{
  private String password="";

  public String getPassword(){
     return password;
  };

  public void setPassword(){
     ...//реализация метода
  };

  public void checkPassword(String pass)
    throws WrongPasswordException {
     if(!pass.equals(password))
        throw new WrongPasswordException();
  };
}
```

При вызове метода `checkPassword` в случае неправильного пароля, переданного в качестве параметра, возбуждается исключительная ситуация. Следует обратить внимание, что сравнение `pass!=password` всегда будет давать `true`, так как строки сравниваются как объекты. То есть при сравнении `"=="` проверяется идентичность адресов в памяти, а не содержание строк.

Еще один момент, на котором следует остановиться: не используйте *возбуждение исключительных ситуаций* для нормального режима функционирования программы! Не используйте его вместо блока `else` в операторе `if`! Возбуждение исключения выполняется намного дольше, потребляет много ресурсов и при неудачном использовании только ухудшает программу. Например, в нашем случае имело бы смысл при неправильном вводе пароля предусмотреть возможность еще двух попыток ввода в обычном режиме – и только после третьей неудачной попытки возбуждать исключение.

7.6. Работа с файлами и папками

Концепция работы с файлами в *Java* включает две составляющие:

Работа с файлами и папками с помощью объектов типа `File`.

Обеспечивает работу с именами файлов (проверка существования файла или папки с заданным именем, нахождение абсолютного пути по относительному и наоборот, проверка и установка *атрибутов файлов* и

папок).

Работа *потоками ввода-вывода*.

Обеспечивает работу не только с файлами, но и с памятью, а также различными устройствами ввода-вывода.

Работа с файлами и папками с помощью объектов типа File

Объекты типа `File` могут рассматриваться как абстракции, инкапсулирующие работу с именами файлов и папок. При этом папка рассматривается как разновидность файла, обладающая особыми атрибутами.

Создание объекта типа `File` осуществляется с помощью конструкторов, имеющих следующие варианты:

File("Имя папки")

File("Имя файла")

File("Имя папки","Имя файла").

При этом имена могут быть как короткими (локальными), без указания пути к файлу или папке, так и длинными (абсолютными), с указанием пути. В приведенной далее таблице файлы (папки) ищутся по имени в соответствии с правилами поиска файлов в операционной системе. Для платформы Windows® вместо символа `"\"` в строках, соответствующих путям, должна использоваться последовательность `"\\"` .

Важнейшие файловые операции, инкапсулированные классом `File` :

Таблица 7.3.

Поле или метод	Что содержит или делает
Переменные класса	
`String pathSeparator`	Содержит строку с символом разделителя пути в операционной системе. Это `"/"` в Unix-подобных системах и `"\"` в Windows®.
`char pathSeparatorChar`	Содержит символ разделителя пути в операционной системе. Это `'/'` в Unix-подобных системах и `'\'` в Windows®.
`String separator`	Содержит строку с символом разделителя между именами файлов и файловых масок в операционной системе.
`char separatorChar`	Содержит символ разделителя между именами файлов и файловых масок в операционной системе.
Проверка параметров файла или папки	
`boolean exists()`	Возвращает `true` в случае, когда файл (или папка) с заданным в конструкторе именем существует. Иначе `false` .
`long length()`	Возвращает длину файла в байтах в случае, когда файл с заданным в конструкторе именем существует и не является папкой. Иначе 0L.
`boolean canRead()`	Возвращает `true` в случае, когда файл (или папка) с заданным в конструкторе именем существует и доступен по чтению. Иначе `false` . (В Unix-подобных системах существуют файлы, доступные только по записи). Может возбуждать `SecurityException` .
`boolean setReadOnly()`	Возвращает `true` в случае, когда файл (или папка) с заданным в конструкторе именем существует, и ему удалось установить статус "доступен только по чтению". Иначе `false` .
`boolean canWrite()`	Возвращает `true` в случае, когда файл (или папка) с заданным в конструкторе именем существует и доступен

	по записи. Иначе `false`. (В операционных системах существуют файлы, доступные только по чтению). Может возбуждать `SecurityException`.
`boolean isDirectory()`	Возвращает `true` в случае, когда файл или папка с заданным в конструкторе именем существует и является папкой. Иначе `false`.
`boolean isFile()`	Возвращает `true` в случае, когда файл или папка с заданным в конструкторе именем существует и является файлом. Иначе `false`.
`boolean isHidden()`	Возвращает `true` в случае, когда файл или папка с заданным в конструкторе именем существует и является скрытым. Иначе `false`. В Unix-образных системах скрытыми являются файлы, имена которых начинаются с точки. В Windows® – те, которые имеют атрибут `"hidden"` ("скрытый").
`long lastModified()`	Возвращает время последней модификации файла, если он существует и доступен по чтению. Иначе 0L. Время отсчитывается в миллисекундах, прошедших с 0 часов 1 января 1970 года (по Гринвичу).
`boolean setLastModified(long time)`	Устанавливает время последней модификации файла. Возвращает `true`, если он существует и доступен по записи. Иначе `false`. Время отсчитывается в миллисекундах, прошедших с 0 часов 1 января 1970 года (по Гринвичу).

Путь и имя файла (папки)

`String getName()`	Возвращает короткое имя файла или папки.
`String getParent()`	Возвращает абсолютное имя родительской папки – то есть папки, в которой находится файл (или папка), соответствующий файловому объекту.
`String getAbsolutePath()`	Возвращает абсолютный путь к файлу или папке, включая имя файла. При этом если в имени файла в конструкторе была задана *относительная адресация*, соответствующая часть пути сохраняется в возвращаемой строке.
`String getCanonicalPath()`	Возвращает абсолютный путь к файлу или папке, включая имя файла. При этом если в имени файла в конструкторе была задана *относительная адресация*, соответствующая часть пути заменяется в возвращаемой строке на канонический вариант адресации – без элементов относительной адресации. Возбуждает `IOException`, если канонический путь не может быть построен.
`int compareTo(File f)`	Сравнение имен файлов (папок), сопоставляемых текущему файловому объекту и объекту `f`. Возвращает 0 в случае, когда абсолютные имена файлов (папок) совпадают. Иначе возвращает число, зависящее от разницы в длинах имен и кодов составляющих их символов. Сравнение зависимо от операционной системы – в Unix-образных системах регистр символов имеет значение, в Windows® – не имеет. Соответствие

	понимается абстрактно на уровне имен и путей – самих файлов может не существовать.
`boolean isAbsolute()`	Возвращает `true` в случае, когда адресация к имени файла (папки) текущего файлового объекта является абсолютной. Хотя может содержать элементы относительной адресации, то есть не быть канонической.
`boolean equals(Object obj)`	Возвращает `true` тогда и только тогда, когда текущий объект и параметр `obj` соответствуют одному и тому же файлу (папке) . С учетом правил о путях и регистрах символов, задаваемых операционной системой. Соответствие понимается абстрактно на уровне имен и путей – самих файлов может не существовать.

Создание/уничтожение/переименование файлов и папок

`boolean createNewFile()`	Попытка создания файла или папки по имени, которое было задано в конструкторе объекта. В случае успеха возвращается `true` , иначе `false` . Возбуждает `IOException` , если файл не может быть создан (например, уже существует).
`File createTempFile(String prefix, String suffix)` `File createTempFile(String prefix, String suffix, File folder)`	Метод класса. Обеспечивает создание пустого файла (или папки), задаваемого коротким именем `prefix+suffix` в папке операционной системы, предназначенной для временных файлов. Возвращает ссылку на объект. Префикс должен быть не менее 3 символов. Возбуждает `IOException` , если файл не может быть создан (например, уже существует).
`boolean mkdir()`	Попытка создания папки по имени, которое было задано в конструкторе объекта. Возвращает `true` в случае успешного создания и `false` в других случаях.
`boolean mkdirs()`	Попытка создания папки по имени, которое было задано в конструкторе объекта, причем заодно создаются все папки, заданные в пути, если они не существовали. Возвращает `true` в случае успешного создания и `false` в других случаях.
`boolean delete()`	Попытка удаления файла или папки по имени, которое было задано в конструкторе объекта. Возвращает `true` в случае успешного удаления и `false` в других случаях.
`boolean renameTo(File dest)`	Попытка переименования файла или папки с имени, которое было задано в конструкторе объекта, на новое, задаваемое параметром `dest` . Возвращает `true` в случае успешного переименования и `false` в других случаях.

Создание нового файлового объекта с помощью имеющегося

`File getAbsoluteFile()`	Создание нового файлового объекта по абсолютному пути, соответствующему текущему файловому объекту.
`File getCanonicalFile()`	Создание нового файлового объекта по каноническому пути, соответствующему текущему файловому объекту.

	Возбуждает `IOException`, если канонический путь не может быть построен.
`File getParentFile()`	Создание нового файлового объекта по абсолютному пути, соответствующему родительской папке для текущего файлового объекта.

Списки папок и файлов

`String[] list()` `String[] list(FilenameFilter filter)`	Возвращает массив строк (список) коротких имен находящихся в папке файлов и папок. Имена элементов, находящихся во *вложенных папках*, не показываются. Если файловый объект не соответствует существующей папке, возвращает `null`. При наличии фильтра возвращаются только те имена, которые соответствуют маске фильтра.
`File[] listFiles()` `File[] listFiles(FilenameFilter filter)`	Возвращает массив файловых объектов, соответствующих находящимся в папке файлов и папок. Элементы, находящиеся во *вложенных папках*, не учитываются. Если текущий файловый объект не соответствует существующей папке, возвращает `null`. При наличии фильтра возвращаются объекты только для тех имен, которые соответствуют маске фильтра.
`File[] listRoots()`	Возвращает массив файловых объектов, соответствующих возможным на данном компьютере корневым папкам. В Unix это папка "/", в Windows® – корневые папки всех возможных дисков.

Пример работы с файловыми объектами:

```
File f1=new File(".."); // "." , "/" , "C:/../"
System.out.println("getAbsolutePath(): "+f1.getAbsolutePath());

try{
System.out.println("getCanonicalPath(): "+f1.getCanonicalPath());
}
catch(Exception e){
System.out.println("Исключение от getCanonicalPath() ");
};

System.out.println("exists(): "+f1.exists());
System.out.println("canRead(): "+f1.canRead());
System.out.println("canWrite(): "+f1.canWrite());
```

Выбор файлов и папок с помощью файлового диалога

При работе с файлами в подавляющем большинстве приложений требуется вызов файлового диалога. В нашем приложении из *палитры компонентов* (правое верхнее окно среды разработки) перетащим на *экранную форму* `JLabel` ("Метка") – первый компонент в палитре *Swing*. А затем повторим эту операцию еще раз. В первой метке мы будем показывать имя выбранного файла, а во второй – путь к этому файлу.

Для того, чтобы вызвать файловый диалог, назначим обработчик события пункту файлового меню **openMenuItem** ("Открыть...") – подузел **[JFrame]/menuBar[JMenu]/fileMenu[JMenu]/openMenuItem[JMenuItem]**. Двойной щелчок по узлу **openMenuItem** приводит к автоматическому созданию заготовки обработчика `openMenuItemActionPerformed` и открытию редактора исходного кода.

Сначала мы создаем в приложении объект типа `JFileChooser`, соответствующий файловому диалогу. Если в начале записать `import javax.swing.*`, что желательно, то в соответствующих местах кода не

потребуется квалификация `javax.swing`. Но в данном примере импорт не сделан намеренно для того, чтобы было видно, где используются классы данного пакета.

```
javax.swing.JFileChooser fileChooser=new javax.swing.JFileChooser();
```

Добавим в обработчик необходимый код:

```
private void openMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
  if(fileChooser.showOpenDialog(null)!= fileChooser.APPROVE_OPTION){
      System.out.println("Отказались от выбора");
      return;
  };
  System.out.println("Нажали Open");
  jLabel1.setText(fileChooser.getSelectedFile().getName());
  jLabel2.setText(fileChooser.getSelectedFile().getParent());
}
```

В первой строке обработчика вызывается метод `fileChooser.showOpenDialog(openMenuItem)`. Он показывает диалог на экране. В качестве параметра должен быть задан родительский компонент – в этом качестве мы используем пункт меню `openMenuItem`. Сравнение с *переменной класса* `APPROVE_OPTION` позволяет выяснить, была ли выбрана кнопка **Open** – "Открыть".

Следует обратить внимание на характерный прием – выход из подпрограммы с помощью оператора `return` в случае, когда не был осуществлен выбор файла. Неопытные программисты написали бы данный фрагмент кода таким образом:

```
if(fileChooser.showOpenDialog(openMenuItem)== fileChooser.APPROVE_OPTION){
  System.out.println("Нажали Open");
  jLabel1.setText(fileChooser.getSelectedFile().getName());
  jLabel2.setText(fileChooser.getSelectedFile().getParent());
}
else
  System.out.println("Отказались от выбора");
```

На первый взгляд принципиальной разницы нет. Но при усовершенствовании программы код, соответствующий выбору файла с помощью диалога, заметно разрастется, а код, соответствующий отказу от выбора, останется тем же. В результате практически весь код обработчика во втором варианте кода окажется вложенным в оператор `if` – а это может быть несколько страниц кода. В таком коде трудно разбираться. В первом варианте оператор `if` обладает небольшой областью действия, что позволяет легко разобраться с относящимся к нему кодом.

Можно было бы перенести строку

```
javax.swing.JFileChooser fileChooser=new javax.swing.JFileChooser()
```

в обработчик события. В этом случае при каждом нажатии пункта меню **"Файл/Открыть…"** создавался бы новый объект-диалог. Такой код был бы работоспособен. Но создание диалога является относительно долгим процессом, требующим большого количества ресурсов операционной системы. Поэтому лучше создавать в приложении глобальную переменную, которой назначен диалог. Помимо прочего это позволяет в повторно открываемом диалоге оказываться в той же папке, где происходил последний выбор файла.

Перед вызовом диалога можно программно установить папку, в которой он будет открываться:

```
File folder=…;
fileChooser.setCurrentDirectory(folder);
```

Диалог сохранения файла открывается аналогичным образом – `showSaveDialog`.

Практически всегда при использовании файловых диалогов требуется задавать фильтр, по которому просматриваются файлы. В подавляющем большинстве случаев требуется определенное расширение в имени файла. К сожалению, компонент `JFileChooser` до сих пор не поддерживает встроенной возможности настраивать фильтр – для этого требуется создание специального класса. Приведем пример простейшего такого класса:

```
package java_gui_example;
import java.io.*;

public class SimpleFileFilter extends javax.swing.filechooser.FileFilter {
  String ext;
  SimpleFileFilter(String ext){
  this.ext=ext;
  }

  public boolean accept(File f){
    if(f==null)
      return false;
    if(f.isDirectory()){
      return true ;
    }
    else
      return (f.getName().endsWith(ext));
  }

  /**
   * Описание фильтра, возникающее в строке фильтра
   * @see FileView#getName
   */
  public String getDescription(){
      return "Text files (.txt)";
  }
}
```

Использование приведенного класса следующее: задается глобальная переменная

```
javax.swing.filechooser.FileFilter fileFilter=new SimpleFileFilter(".txt");
```

После чего она используется в обработчике события:

```
fileChooser.addChoosableFileFilter(fileFilter);
```

Данный оператор добавляет фильтр в выпадающий список возможных фильтров (масок) файлового диалога. Если добавляется уже существующий фильтр, операция добавления игнорируется.

По умолчанию показывается последний из добавленных фильтров. Если требуется показать другой фильтр, который уже был добавлен в список фильтров диалога, требуется вызвать оператор

```
fileChooser.setFileFilter(fileFilter);
```

Он делает добавленный фильтр текущим, то есть видимым при открытии диалога.

Кроме стандартных диалогов открытия и сохранения файлов имеется возможность вызова диалога с дополнительными программно задаваемыми элементами – с помощью метода `showCustomDialog` .

Также имеется возможность выбирать несколько файлов. Для этого до вызова диалога требуется задать разрешение на такой выбор:

```
fileChooser.setMultiSelectionEnabled(true);
```

Получение массива выбранных файлов после вызова диалога идет следующим образом:

```
java.io.File[] files = fileChooser.getSelectedFiles();
if (files != null && files.length > 0) {
  String filenames = "";
```

```
    for (int i=0; i<files.length; i++) {
  filenames = filenames + "\n" + files[i].getPath();
    }
  }
```

Имеется возможность выбора папки (директории), а не файла. В этом случае следует задать режим, когда позволяется выбирать только папки

```
  fileChooser.setFileSelectionMode(fileChooser.DIRECTORIES_ONLY);
```

либо и файлы, и папки

```
  fileChooser.setFileSelectionMode(fileChooser.FILES_AND_DIRECTORIES);
```

Возврат в обычный режим:

```
  fileChooser.setFileSelectionMode(fileChooser.FILES_ONLY);
```

Для того, чтобы в выбранной папке просмотреть список файлов в папке, с которой связана переменная `File folder` , используется вызов вида

```
  String[] filenames= folder.list(filter);
```

Получается массив строк с короткими именами файлов. Используется переменная `filter` , тип которой является классом, реализующим интерфейс `java.io.FilenameFilter` . О том, что такое интерфейсы, будет рассказано в отдельном разделе. Пример простейшего такого класса `SimpleFilenameFilter` :

```
  package java_gui_example;
  import java.io.*;
  public class SimpleFilenameFilter implements FilenameFilter{
      String ext;
      public SimpleFilenameFilter(String ext) {
        this.ext=ext;
      }

      public boolean accept(File dir,String fileName){
         return ext==""||fileName.endsWith(ext);
      }
  }
```

Пример с показом файлов, содержащихся в выбранной папке, в компонент `jTextArea1` (текстовая область) типа `JTextArea` , позволяющий показывать произвольное число строк:

```
  String[] filenamesArray;
  File folder=fileChooser.getSelectedFile();
  SimpleFilenameFilter filter=new SimpleFilenameFilter("");
  String filenames = "";
  if(folder.isDirectory()){
      filenamesArray=folder.list(filter);
      for (int i=0; i<filenamesArray.length; i++) {
          filenames = filenames + "\n" + filenamesArray[i];
      };
      jTextArea1.setText(filenames);
  }
```

Аналогичный пример вывода выбранных в диалоге имен в режиме "мультиселект", позволяющем отмечать несколько файлов и/или папок:

```
java.io.File[] files = fileChooser.getSelectedFiles();
if (files != null && files.length > 0) {
    String filenames = "";
    for (int i=0; i<files.length; i++) {
        filenames = filenames + "\n" + files[i].getPath();
    };
    jTextArea1.setText(filenames);
};
```

Работа с потоками ввода-вывода

Стандартные классы Java, инкапсулирующие работу с данными (оболочечные классы для числовых данных, классы `String` и `StringBuffer`, массивы и т.д.), не обеспечивают поддержку чтения этих данных из файлов или запись их в файл. Вместо этого используется весьма гибкая и современная, но не очень простая технология использования *потоков* (Streams).

Поток представляет накапливающуюся последовательность данных, поступающих из какого-то источника. Порция данных может быть считана из потока, при этом она из потока изымается. В потоке действует принцип очереди – "первым вошел, первым вышел".

В качестве источника данных потока может быть использован как стационарный источник данных (файл, массив, строка), так и динамический – другой поток. При этом в ряде случаев выход одного потока может служить входом другого. Поток можно представить себе как трубу, через которую перекачиваются данные, причем часто в таких "трубах" происходит обработка данных. Например, поток шифрования шифрует данные, полученные на входе, и при считывании из потока передает их в таком виде на выход. А поток *архивации* сжимает по соответствующему алгоритму входные данные и передает их на выход. "Трубы" потоков можно соединять друг с другом – выход одного со входом другого. Для этого в качестве параметра конструктора потока задается имя переменной, связанной с потоком – источником данных для создаваемого потока.

Буферизуемые потоки имеют хранящийся в памяти промежуточный буфер, из которого считываются выходные данные потока. Наличие такого буфера позволяет повысить производительность операций ввода-вывода, а также осуществлять дополнительные операции – устанавливать метки (маркеры) для какого-либо элемента, находящегося в буфере потока и даже делать возврат считанных элементов в поток (в пределах буфера).

Абстрактный класс `InputStream` ("входной поток") инкапсулирует модель входных потоков, позволяющих считывать из них данные. *Абстрактный класс* `OutputStream` ("выходной поток") – модель выходных потоков, позволяющих записывать в них данные. *Абстрактные методы* этих классов реализованы в классах-потомках.

Таблица 7.4. Методы класса InputStream

Метод	Что делает
`int available()`	Текущее количество байт, доступных для чтения из потока.
`int read()`	Читает один байт из потока и возвращает его значение в виде целого, лежащего в диапазоне от 0 до 255. При достижении конца потока возвращает -1.
`int read(byte[] b)` `int read(byte[] b, int offset, int count)`	Пытается прочесть `b.length` байт из потока в массив `b`. Возвращает число реально прочитанных байт. После достижения конца потока последующие считывания возвращают -1.
`long skip(long count)`	Попытка пропускать (игнорировать) `count` байт из потока. count ≤ 0, пропуска байт нет. Возвращается реально пропущенное число байт. Если это значение ≤ 0, пропуска байт не было.
`boolean markSupported()`	Возвращает `true` в случае, когда поток поддерживает операции `mark` и `reset`, иначе – `false`.
`void mark(int limit)`	Ставит метку в текущей позиции начала потока. Используется для последующего вызова метода `reset`, с помощью которого

	считанные после установки метки данные возвращаются обратно в поток. Эту операцию поддерживают не все потоки. См. метод `markSupported`.
`void reset()`	Восстанавливает предшествующее состояние данных в начале потока, возвращая указатель начала потока на помеченный до того меткой элемент. То есть считанные после установки метки данные возвращаются обратно в поток. Попытка вызова при отсутствии метки или выходе ее за пределы лимита приводит к возбуждению исключения `IOException`. Эту операцию поддерживают не все потоки. См. методы `markSupported` и `mark`.
`void close()`	Закрытие потока. Последующие попытки чтения из этого потока приводят к возбуждению исключения `IOException`.

Все методы класса `InputStream`, кроме `markSupported` и `mark`, возбуждают исключение `IOException` – оно возникает при ошибке чтения данных.

Таблица 7.5. Методы класса OutputStream

Метод	Что делает
`void write(int b)`	Записывает один байт в поток. Благодаря использованию типа `int` можно использовать в качестве параметра целочисленное выражение без приведения его к типу `byte`. Напомним, что в выражениях "короткие" целые типы автоматически преобразуются к типу `int`.
`void write(byte[] b)`	Записывает в поток массив байт. Если заданы параметры `offset` и `count`, записывается не весь массив, а `count` байт начиная с индекса `offset`.
`void write(byte[] b, int offset, int count)`	
`void flush()`	Форсирует вывод данных из выходного буфера и его очистку. Необходимость этой операции связана с тем, что в большинстве случаев данные уходят "во внешний мир" на запись не каждый раз после вызова `write`, а только после заполнения выходного буфера. Таким образом, если операции записи разделены паузой (большим интервалом времени), и требуется, чтобы уход данных из выходного буфера совершался своевременно, после последнего вызова оператора `write`, предшествующего паузе, надо вызывать оператор *`flush`*.
`void close()`	Закрытие потока. Последующие попытки записи в этот поток приводят к возбуждению исключения `IOException`.

Все методы этого класса возбуждают `IOException` в случае ошибки записи.

Не все классы потоков являются потомками `InputStream/OutputStream`. Для чтения строк (в виде массива символов) используются потомки *абстрактного класса* `java.io.Reader` ("читатель"). В частности, для чтения из файла – класс `FileReader`. Аналогично, для записи строк используются классы – потомки *абстрактного класса* `java.io.Writer` ("писатель"). В частности, для записи массива символов в файл – класс `FileWriter`.

Имеется еще один важный класс для работы с файлами, не входящий в иерархии `InputStream/OutputStream` и `Reader/ Writer`. Это класс `RandomAccessFile` ("файл с произвольным доступом"), предназначенный для чтения и записи данных в произвольном месте файла. Такой

файл с точки зрения класса `RandomAccessFile` представляет массив байт, сохраненных на внешнем носителе. Класс `RandomAccessFile` не является абстрактным, поэтому можно создавать его экземпляры.

Все операции чтения и записи начинаются с позиции, задаваемой файловым указателем перед началом операции. После каждой такой операции файловый указатель сдвигается к концу файла на число позиций (байт), соответствующее типу считанных данных. В случае, когда во время записи файловый указатель выходит за конец файла, размер файла автоматически увеличивается.

Таблица 7.6. Важнейшие методы класса RandomAccessFile

Метод	Что делает
Общие параметры файла	
`long getFilePointer()`	Возвращает значение файлового указателя – текущую позицию в файле считывающей/записывающей головки. Индексация начинается с нуля и дается в числе байт.
`long length()`	Длина файла (в байтах).
`void setLength(long newLength)`	Изменение длины файла – она устанавливается равной `newLength`. Если `newLength` меньше текущей длины, файл укорачивается путем отбрасывания его конца. При этом если файловый указатель находился в отбрасываемой части, он устанавливается в конец файла, иначе его позиция не меняется. Если `newLength` больше текущей длины, файл увеличивает размер путем добавления в конец нового пространства, содержимое которого не гарантируется.
`void close()`	Закрытие потока. Последующие попытки доступа к этому потоку приводят к возбуждению исключения `IOException`.
Побайтные операции установки позиции	
`void seek(long pos)`	Установка файлового указателя в позицию *pos*.
`int skipBytes(int n)`	Пропуск `n` байт – передвижение считывающей/записывающей головки на `n` байт. Если $n > 0$, то вперед, к концу файла. Если $n \leq 0$, то позиция головки не меняется. Возвращается число реально пропущенных байт. Оно может быть меньше n – например, из-за достижения конца файла.
Побайтные операции чтения	
`int read()`	Читает один байт из потока и возвращает его значение в виде целого, лежащего в диапазоне от 0 до 255. При достижении конца потока возвращает -1.
`int read(byte[] b)`	Пытается прочесть `b.length` байт из потока в массив `b`. Возвращает число реально прочитанных байт. После достижения конца потока последующие считывания возвращают -1.
`int read(byte[] b, int offset, int count)`	Пытается прочесть `count` байт из потока в массив `b`, записывая их начиная с позиции `offset`. Возвращает число реально прочитанных байт. После достижения конца потока последующие считывания возвращают -1.
`void readFully(byte[] b)`	Считывает из файла `b.length` байт, начиная с текущей позиции файлового указателя, и заполняет весь буферный массив `b`.
`void readFully(byte[] b, int offset, int count)`	Считывает из файла в массив `count` байт, записывая их начиная с байта, имеющего позицию `offset`.

Побайтные операции записи

`void write(int b)`	*Запись в файл* одного байта `b`. То же, что `writeByte(b)`.
`void write(byte[] b)`	*Запись в файл* всего массива байт `b`, т.е. `b.length` байт.
`void write(byte[] b, int offset, int count)`	*Запись в файл* `count` байт массива `b` начиная с байта массива, имеющего индекс `offset`.

Чтение одиночных значений примитивного типа

`boolean readBoolean()`	Считывает значение `boolean`
`byte readByte()`	Считывает значение `byte`
`short readShort()`	Считывает значение `short`
`int readInt()`	Считывает значение `int`
`long readLong()`	Считывает значение `long`
`int readUnsignedByte()`	Считывает значение типа "беззнаковый байт"(тип отсутствует в Java) и преобразует в значение типа `int`
`int readUnsignedShort()`	Считывает значение типа "беззнаковое короткое целое"(тип отсутствует в Java) и преобразует в значение типа `int`
`char readChar()`	Считывает значение `char`
`float readFloat()`	Считывает значение `float`
`double readDouble()`	Считывает значение `double`

Запись одиночных значений примитивного типа

`void writeBoolean(boolean v)`	*Запись в файл* значение `boolean`
`void writeByte(int v)`	*Запись в файл* значение `byte`. Благодаря использованию типа `int` можно использовать в качестве параметра целочисленное выражение без приведения его к типу `byte`.
`void writeShort(int v)`	*Запись в файл* значение `short`. Благодаря использованию типа `int` можно использовать в качестве параметра целочисленное выражение без приведения его к типу `short`.
`void writeInt(int v)`	*Запись в файл* значение `int`
`void writeLong(long v)`	*Запись в файл* значение `long`
`void writeChar(int v)`	*Запись в файл* значение `char`. Благодаря использованию типа `int` можно использовать в качестве параметра целочисленное выражение без приведения его к типу `char`.
`void writeFloat(float v)`	*Запись в файл* значение `float`
`void writeDouble(double v)`	*Запись в файл* значение `double`

Чтение отдельных строк

`String readLine()`	Считывает из файла строку символов от текущей позиции файлового указателя до места переноса на новую строку (символ *возврата каретки* или последовательность `"\n"`) или конца файла. При этом старшие байты получившихся в строке символов UNICODE оказываются равными нулю, т.е. поддерживается только часть кодовой таблицы. Метод

	полезен для считывания текста, записанного в кодировке ANSI.
String readUTF	Считывает из файла строку символов в кодировке UTF-8 от текущей позиции файлового указателя. Число последующих считываемых байт строки в кодировке UTF-8 задается первыми двумя считанными байтами.

Запись отдельных строк

void writeBytes(String s)	*Запись в файл* строки s как последовательности байт, соответствующих символам строки. При этом каждому символу соответствует один записываемый байт, так как у каждого символа UNICODE отбрасывается старший из двух байт (записывается младший байт). Используется для записи символов в кодировке ANSI. См. также метод writeChars , в котором записываются оба байта.
void writeChars(String s)	*Запись в файл* строки s как последовательности байт, соответствующих символам строки. При этом каждому символу UNICODE соответствует два записываемых байта.
void writeUTF(String s)	*Запись в файл* строки s как последовательности байт, соответствующих символам строки в кодировке UTF-8.

Все методы класса RandomAccessFile , кроме getChannel() , возбуждают исключение IOException – оно возникает при ошибке записи или ошибке доступа к данным.

В конструкторе класса требуется указать два параметра. Первый – имя файла или *файловую переменную*, второй – строку с модой доступа к файлу. Имеются следующие варианты:

"r"- от read,- "только читать"

"rw"- от read and write,- "читать и писать"

"rws"- от read and write synchronously,- "читать и писать с поддержкой синхронизации" (см. далее раздел про потоки, Threads)

"rwd"- от read and write to device,- "читать и писать с поддержкой синхронизации устройства" (см. далее раздел про потоки, Threads)

Например:

```
java.io.RandomAccessFile rf1=new java.io.RandomAccessFile("q.txt","r");
java.io.RandomAccessFile rf2=new java.io.RandomAccessFile(file,"rw");
```

Рассмотрим примеры, иллюстрирующие работу с потоками.

Пример чтения текста из файла:

```
File file;
javax.swing.JFileChooser fileChooser=new javax.swing.JFileChooser();
javax.swing.filechooser.FileFilter fileFilter=new SimpleFileFilter(".txt");

private void openMenuItemActionPerformed(java.awt.event.ActionEvent evt) {

    fileChooser.addChoosableFileFilter(fileFilter);
    if(fileChooser.showOpenDialog(null)!=fileChooser.APPROVE_OPTION){
        return;//Нажали Cancel
    };
    file = fileChooser.getSelectedFile();
    try{
        InputStream fileInpStream=new FileInputStream(file);
        int size=fileInpStream.available();
        fileInpStream.close();
```

```
            char[] buff=new char[size];
            Reader fileReadStream=new FileReader(file);
            int count=fileReadStream.read(buff);
            jTextArea1.setText(String.copyValueOf(buff));
            javax.swing.JOptionPane.showMessageDialog(null,
                    "Прочитано "+ count+" байт");

            fileReadStream.close();
        } catch(Exception e){
            javax.swing.JOptionPane.showMessageDialog(null,
                    "Ошибка чтения из файла \n"+file.getAbsolutePath());
        }
    }
```

Переменной `fileInpStream`, которая будет использована для работы потока `FileInputStream`, мы задаем тип `InputStream`. Это очень характерный прием при работе с потоками, позволяющий при необходимости заменять в дальнейшем входной поток с `FileInputStream` на любой другой без изменения последующего кода. После выбора имени текстового файла с помощью файлового диалога мы создаем файловый поток:

```
fileInpStream=new FileInputStream(file);
```

Функция `fileInpStream.available()` возвращает число байт, которые можно считать из файла. После ее использования поток `fileInpStream` нам больше не нужен, и мы его закрываем. Поток `FileReader` не поддерживает метод `available()`, поэтому нам пришлось использовать поток типа `FileInputStream`.

Массив `buff` используется в качестве буфера, куда мы считываем весь файл.

Если требуется читать файлы большого размера, используют буфер фиксированной длины и в цикле считывают данные из файла блоками, равными длине буфера, до тех пор, пока число считанных байт не окажется меньше размера буфера. Это означает, что мы дошли до конца файла. В таком случае выполнение цикла следует прекратить. При такой организации программы нет необходимости вызывать метод `available()` и использовать поток типа `FileInputStream`.

В операторе

```
Reader fileReadStream=new FileReader(file);
```

используется тип *`Reader`*, а не `FileReader`, по той же причине, что до этого мы использовали `InputStream`, а не `FileInputStream`.

Построчное чтение из файла осуществляется аналогично, но содержимое блока `try` следует заменить на следующий код:

```
FileReader filReadStream=new FileReader(file);
BufferedReader bufferedIn=new BufferedReader(filReadStream);
String s="",tmpS="";
while((tmpS=bufferedIn.readLine())!=null)
    s+=tmpS+"\n";
jTextArea1.setText(s);
bufferedIn.close();
```

Пример записи текста в файл очень похож на пример чтения, но гораздо проще, так как не требуется вводить промежуточные буферы и потоки:

```
private void saveAsMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    fileChooser.addChoosableFileFilter(fileFilter);
    if(fileChooser.showSaveDialog(null)!=fileChooser.APPROVE_OPTION){
        return;//Нажали Cancel
    };
    file = fileChooser.getSelectedFile();
```

```
        try{
            Writer filWriteStream=new FileWriter(file);
            filWriteStream.write(jTextArea1.getText() );
            filWriteStream.close();
        } catch(Exception e){
            javax.swing.JOptionPane.showMessageDialog(null,
                       "Ошибка записи в файл \n"+file.getAbsolutePath());
        }
    }
```

7.7. Типы-перечисления (enum)

Иногда требуется использовать элементы, которые не являются ни числами, ни строками, но ведут себя как имена элементов и одновременно обладают порядковыми номерами. Например, названия месяцев или дней недели. В этих случаях используют *перечисления*. Для задания типа какого-либо перечисления следует написать *зарезервированное слово* `enum` (сокращение от *enumeration* – "*перечисление*"), после которого имя задаваемого типа, а затем в фигурных скобках через запятую элементы перечисления. В качестве элементов можно использовать любые простые идентификаторы (не содержащие квалификаторов вида *имя1.имя2*).

Тип-*перечисление* обязан быть глобальным – он может задаваться либо на уровне пакета, либо в каком-либо классе. Но его нельзя задавать внутри какого-либо метода. Элементы перечисления могут иметь любые имена, в том числе совпадающие в разных перечислениях или совпадающие с именами классов или их членов – каждое *перечисление* имеет свое собственное *пространство имен*. *Доступ* к элементу перечисления осуществляется с квалификацией именем типа-перечисления:

ИмяТипа.имяЭлемента

У каждого элемента перечисления имеется порядковый номер, соответствующий его положению в наборе – *нумерация* начинается с нуля. Поэтому первый элемент имеет номер 0, второй элемент – номер 1, и так далее. Имеется *функция* `ordinal()` , возвращающая порядковый номер элемета в перечислении. Также имеется *функция* `compareTo` , позволяющая сравнивать два элемента перечисления – она возвращает разницу в их порядковых номерах.

Строковое *представление* значения можно получить с помощью функции `name()` . Преобразование из строки в *значение* типа "*перечисление*" осуществляется с помощью функции класса `valueOf` , в которую передается строковое *представление* значения.

Если требуется рассматривать элементы перечисления как *массив*, можно воспользоваться функцией `values()` – она возвращает *массив* элементов, к которым можно обращаться по индексу. Формат вызова функции такой: *ИмяТипа.values()*

Для примера зададим типы-перечисления Monthes ("месяцы") и *Spring* ("весна"), соответствующие различным наборам месяцев:

```
enum Monthes {jan,feb,mar,apr,may,jun,jul,aug,sept,oct,nov,dec};
enum Spring { march, apr, may };
```

Названия месяцев мы намеренно пишем со строчной буквы для того, чтобы было понятно, что это *идентификаторы переменных*, а не типы. А имя марта написано по-разному в типах `Monthes` и `Spring` для того, чтобы показать независимость их пространств имен. Объявление переменных типа "*перечисление*" делается так же, как для всех остальных типов, при этом переменные могут быть как неинициализированы, так и инициализированы при задании:

`public Monthes m1 ,m2=Monthes.mar, m3;` – при задании в классе общедоступных полей `m1` , `m2` и `m3` ,

`Spring spr1=Spring.apr, spr2;` – при задании в методе локальной переменной или задании в классе поля `spr1` с пакетным уровнем доступа.

После чего возможны следующие *операторы*:

```
spr2=spr1;
spr1=Spring.may;
System.out.println("Результат сравнения="+spr2.compareTo(Spring.march));
```

После выполнения этих операторов в консольное окно будет выведен текст

```
Результат сравнения=1 ,
```

поскольку в переменной `spr2` окажется *значение* `Spring.apr`, порядковый номер которого на 1 больше, чем у значения `Spring.march`, с которым идет сравнение.

Пусть в переменной `spr2` хранится *значение* `Spring.may` . Порядковый номер значения, хранящегося в переменной, можно получить с помощью вызова `spr2.ordinal()`. Он возвратит число 2, так как `may` – третий элемент перечисления (сдвиг на 1 получается из-за того, что *нумерация* начинается с нуля).

Строковое *представление* значения, хранящегося в переменной `spr2` , можно получить с помощью вызова `spr2.name()`. Он возвратит строку `"may"` – имя типа в возвращаемое *значение* не входит.

Если *переменная* типа "*перечисление*" не инициализирована, в ней хранится *значение* `null` . Поэтому вызов

```
System.out.println("spr2="+spr2);
```

осуществленный до присваивания переменной `spr2` значения возвратит строку

```
spr2=null
```

А вот попытки вызовов `spr2.ordinal()` или `spr2.name()` приведут к возникновению ошибки (исключительной ситуации) с диагностикой

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
```

Получение значения типа `Spring` по номеру, хранящемуся в переменной `i` , осуществляется так:

```
spr1=Spring.values()[i];
```

Преобразование из строки в *значение* типа `Spring` будет выглядеть так:

```
spr1=Spring.valueOf("march");
```

Краткие итоги

В Java *массивы являются объектами*, но особого рода – их объявление отличается от объявления других видов объектов. Переменная типа массив является ссылочной. Массивы Java являются динамическими – в любой момент им можно задать новую длину.

Двумерный массив представляет собой массив ячеек, каждая из которых имеет тип "*одномерный массив*". Трехмерный – массив двумерных массивов. Соответствующим образом они и задаются.

Для копирования массивов лучше использовать метод `System.arraycopy` . Быстрое заполнение массива одинаковыми значениями может осуществляться методом `Arrays.fill` – класс `Arrays` расположен в пакете `java.util` Поэлементное сравнение массивов следует выполнять с помощью метода `Arrays.equals` (сравнение на равенство содержимого массивов) либо `Arrays.deepEquals` (глубокое сравнение, то есть на равенство содержимого, а не ссылок – на произвольном уровне вложенности). Сортировка (упорядочение по значениям) массива производится методом `Arrays.sort` .

Коллекции (Collections) – "умные" массивы с динамически изменяемой длиной, поддерживающие ряд важных дополнительных операций по сравнению с массивами. Доступ к элементам коллекции в общем случае не может осуществляться по индексу, так как не все коллекции поддерживают индексацию элементов. Эту функцию осуществляют с помощью специального объекта – *итератора* (*iterator*). У каждой коллекции имеется свой *итератор* который умеет с ней работать.

Класс `String` инкапсулирует действия со строками. Объект типа `String` – строка, состоящая из произвольного числа символов, от 0 до $2 * 10^9$. Литерные константы типа `String` представляют собой последовательности символов, заключенные в двойные кавычки. В классе `Object` имеется метод `toString()` , обеспечивающий строковое представление любого объекта.

Строки типа `String` являются неизменяемыми объектами – при каждом изменении содержимого строки создается новый объект-строка. Для того чтобы сделать работу с многочисленными присваиваниями более эффективной, используются классы `StringBuffer` и `StringBuilder` .

Вывод графики осуществляется с помощью методов объекта типа `java.awt.Graphics` . Для того, чтобы результаты вывода не пропадали, в *классе приложения* требуется переопределить метод `paint` , вызываемый при отрисовке. А также обработчик события `ComponentResized` .

Исключительные ситуации в Java являются объектами. Их типы являются классами-потомками *объектного типа* `Throwable` . От `Throwable` наследуются классы `Error` ("Ошибка") и `Exception` ("Исключение"). Экземплярами класса `Error` являются непроверяемые исключительные ситуации, которые невозможно перехватить в блоках `catch` . Экземплярами класса `Exception` и его потомков

являются проверяемые исключительные ситуации. Кроме одного потомка – класса RuntimeException (и его потомков).

Непроверяемые исключения генерируются и обрабатываются системой автоматически – как правило, приводя к завершению приложения. При этом их типы нигде не указываются, и слово throws в заголовке метода указывать не надо. Если же в теле реализуемого метода используется вызов метода, который может возбуждать исключительную ситуацию, и это исключение не перехватывается, в заголовке реализуемого метода требуется указывать соответствующий тип возбуждаемого исключения.

Объекты типа File обеспечивают работу с именами файлов и папок (проверка существования файла или папки с заданным именем, нахождение абсолютного пути по относительному и наоборот, проверка и установка *атрибутов файлов* и папок).

При работе с файлами в подавляющем большинстве приложений требуется вызов файлового диалога JFileChooser (пакет javax.*swing*).

Потоки ввода-вывода обеспечивают работу не только с файлами, но и с памятью, а также различными устройствами ввода-вывода. Соответствующие классы расположены в пакете java.io. *Абстрактный класс* InputStream ("входной поток") инкапсулирует модель входных потоков, позволяющих считывать из них данные. *Абстрактный класс* OutputStream ("выходной поток") – модель выходных потоков, позволяющих записывать в них данные.

Для чтения строк (в виде массива символов) используются потомки *абстрактного класса* *Reader* ("читатель"). В частности, для чтения из файла – класс FileReader . Аналогично, для записи строк используются классы- потомки *абстрактного класса* Writer ("писатель"). В частности, для записи массива символов в файл – класс FileWriter .

Имеется еще один важный класс для работы с файлами – RandomAccessFile ("файл с произвольным доступом"), предназначенный для чтения и записи данных в произвольном месте файла. Такой файл с точки зрения класса RandomAccessFile представляет массив байт, сохраненных на внешнем носителе. Класс RandomAccessFile не является абстрактным, поэтому можно создавать его экземпляры.

Задания

Написать пример с графическим пользовательским интерфейсом для записи численных данных в файл и считывания их из файла с помощью классов FileInputStream/FileOutputStream (потомков InputStream/OutputStream). Использовать диалоги выбора файлов.

Написать пример с графическим пользовательским интерфейсом для записи численных данных в файл и считывания их из файла с помощью класса RandomAccessFile . Использовать диалоги выбора файлов.

Написать пример с графическим пользовательским интерфейсом для записи строковых данных в файл из компонента JTextArea и считывания их из файла с помощью класса RandomAccessFile . Использовать диалоги выбора файлов. Проверить работу с русским языком при записи и чтении текста в различных кодировках (ANSI, UNICODE,UTF-8).

Написать пример сложения, вычитания, умножения матриц задаваемых пользователем размеров. Для визуального представления матриц использовать компоненты JTable . Действия проводить с помощью массивов чисел типа double .

Усложнить пример, добавив возможность сохранения каждой из трех матриц в текстовый файл, и загрузки первых двух матриц из текстового файла.

Внимание! Если Вы увидите ошибку на нашем сайте, выделите её и нажмите Ctrl+Enter.

© Национальный Открытый Университет "ИНТУИТ", 2022 | www.intuit.ru