**Задание.**

1. Превратить описанный ниже класс **ObjectQueue** в generics-класс (обобщение типа данных).
2. С помощью generics-класса **ObjectQueue** содержимое текстового файла f, разделенное на строки, переписать в текстовый файл g, перенося при этом в конец каждой строки все входящие в нее цифры (с сохранением исходного взаимного порядка, как среди цифр, так и среди остальных литер строки), в самом конце строки через пробел написать количество цифр.

**Пример — очередь объектов**

Пример, который позволит посмотреть, как можно организовать взаимодействие объектов для реализации достаточно распространенной структуры данных — очереди. Т.е. создать класс, который позволит «складывать» туда объекты в заранее неизвестном количестве и «вынимать» объекты из этой очереди. Такая функциональность часто называется FIFO — First In First Out — первый пришел, первый ушел. По сути класс должен иметь три метода:

1. Положить объект (произвольного класса) в очередь — назовем метод **push**
2. Вытащить объект (произвольного класса) из очереди — назовем метод **pull**
3. Получить количество объектов в очереди — назовем его **size**

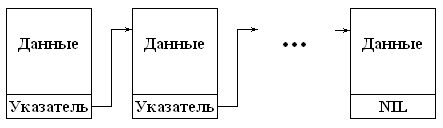
Итак, наш класс **ObjectQueue** будет выглядеть (без реализации, только с методами) вот так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class ObjectQueue  {      public void push(Object obj) {          // Пока реализации нет      }        public Object pull() {          // Пока реализации нет      }        public int size() {          // Пока реализации нет      }  } |

Как видите, можно положить в очередь объект типа **Object**, что означает, что можно положить туда все, что угодно — как вы должны помнить, класс **Object** является «проматерью» всех классов — все классы наследуются в конечном итоге от него (если не прямо от него, то он будет их дедушкой, прадедушкой и т.д.).  
Предлагаю сразу написать маленький класс для тестирования нашей очереди

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class QueueTest  {      public static void main(String[] arg) {          ObjectQueue queue = new ObjectQueue();            for(int i=0; i<10; i++) {              // В данном случае мы складываем в очередь строки              queue.push("Строка:" + i);          }            while(queue.size() > 0) {              // Мы делаем жесткое приведение типа, т.к. знаем, что там лежат строки              String s = (String)queue.pull();              System.out.println(s);              System.out.println("Размер очереди:" + queue.size());          }      }  } |

Можно видеть, что сначала мы помещаем в очередь объект класса **String** (первый цикл от 0 до 10 и потом выбираем все объекты до тех пор, пока размер очереди не станет нулевым. Обратите внимание на то, что делаем приведение типа объекта при его получении.   
Теперь давайте подумаем над реализацией нашей очереди. Большинство книг по структурам данных содержит такое понятие как связный список. Если быть точнее — однонаправленный связный список. Сама идея списка достаточно простая — ее можно посмотреть на рисунке.



Как видите, все достаточно несложно. У нас есть структура, которая содержит два элемента:

1. Хранилище для данных
2. Указатель на следующий элемент

При добавлении нового элемента нам надо создать такую структуру (по сути это объект с двумя полями), поместить в хранилище данных переданный объект и указатель у последнего элемента в списке «указать» на добавляемый объект. Мы как бы «прикрепляем» новый объект к существующей цепочке объектов. Однонаправленная очередь потому, что двигаться по ней можно только в одном направлении — от «головы» к «хвосту».  
Потребуется вспомогательный объект — назовем его **OjectBox** и т.к. он в общем-то никому не нужен, мы можем объявить его внутри нашего класса **ObjectQueue**. Смотрим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | private class ObjectBox {      private Object object;      private ObjectBox next;        public Object getObject() {          return object;      }        public void setObject(Object object) {          this.object = object;      }        public ObjectBox getNext() {          return next;      }        public void setNext(ObjectBox next) {          this.next = next;      }  } |

Как видите — все просто. В поле **object** мы будем помещать сам добавляемый объект, а поле **next** будет указывать на следующий элемент в цепочке.  
Теперь поговорим о классе **ObjectQueue**. Ему необходимо иметь поле, которое указывает на самый первый элемент — нам же надо с чего-то начинать и брать элементы из очереди мы будем как раз с «головы». В принципе одного этого элемента достаточно, но это будет крайне не эффективно. Потому что при добавлении вам придется каждый раз пробегать от «головы» до «хвоста» и уже только после нахождения «хвоста» можно будет добавлять новый элемент. Но сделаем более корректно — введем еще одно поле, которое будет всегда указывать на «хвост» — на последний элемент в очереди. Наличие этого элемента позволяет создавать очень эффективную структуру с очень важной характеристикой — время добавления нового элемента не зависит от размера списка. Очередь может содержать 100 элементов или 100000 — время добавления будет всегда одинаковое. Но у такой структуры есть минус — если вы хотите найти элемент на определенной позиции — например 184-й — то эта операция будет исполняться долго для больших списков. Но вернемся к задаче.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | package edu.javacourse.queue;    public class ObjectQueue  {      // Указатель на первый элемент      private ObjectBox head = null;      // Указатель на последний элемент      private ObjectBox tail = null;      // Поле для хранения размера очереди      private int size = 0;        public void push(Object obj) {          // Сразу создаем вспомогательный объект и помещаем новый элемент в него          ObjectBox ob = new ObjectBox();          ob.setObject(obj);          // Если очередь пустая - в ней еще нет элементов          if (head == null) {              // Теперь наша голова указывает на наш первый элемент              head = ob;          } else {              // Если это не первый элемент, то надо, чтобы последний элемент в очереди              // указывал на вновь прибывший элемент              tail.setNext(ob);          }          // И в любом случае нам надо наш "хвост" переместить на новый элемент          // Если это первый элемент, то "голова" и "хвост" будут указывать на один и тот же элемент          tail = ob;          // Увеличиваем размер нашей очереди          size++;      }        public Object pull() {          // Если у нас нет элементов, то возвращаем null          if (size == 0) {              return null;          }          // Получаем наш объект из вспомогательного класса из "головы"          Object obj = head.getObject();          // Перемещаем "голову" на следующий элемент          head = head.getNext();          // Если это был единственный элемент, то head станет равен null          // и тогда tail (хвост) тоже дожен указать на null.          if (head == null) {              tail = null;          }          // Уменьшаем размер очереди          size--;          // Возвращаем значение          return obj;      }        public int size() {          return size;      }        // Наш вспомогательный класс будет закрыт от посторонних глаз      private class ObjectBox      {          // Поле для хранения объекта          private Object object;          // Поле для указания на следующий элемент в цепочке.          // Если оно равно NULL - значит это последний элемент          private ObjectBox next;            public Object getObject() {              return object;          }            public void setObject(Object object) {              this.object = object;          }            public ObjectBox getNext() {              return next;          }            public void setNext(ObjectBox next) {              this.next = next;          }      }  } |

Класс для тестирования нашей очереди

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package edu.javacourse.queue;    public class QueueTest  {      public static void main(String[] arg) {          ObjectQueue queue = new ObjectQueue();            for(int i=0; i<10; i++) {              // В данном случае мы складываем в очередь строки              queue.push("Строка:" + i);          }            while(queue.size() > 0) {              // Мы делаем жесткое приведение типа, т.к. знаем, что там лежат строки              String s = (String)queue.pull();              System.out.println(s);              System.out.println("Размер очереди:" + queue.size());          }      }  } |

**Добавляем функциональность.**

Добавим удобную функцию — получение элемента по индексу (по номеру в очереди. Эта функция не будет удалять элемент из очереди — она просто вернет элемент. В этом коде увидим, что скорость исполнения здесь зависит от размера очереди — чем он больше, тем функция будет работать дольше. Смотрим код.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102 | package edu.javacourse.queue;    public class ObjectQueue  {      // Указатель на первый элемент      private ObjectBox head = null;      // Указатель на последний элемент      private ObjectBox tail = null;      // Поле для хранения размера очереди      private int size = 0;        public void push(Object obj) {          // Сразу создаем вспомогательный объект и помещаем новый элемент в него          ObjectBox ob = new ObjectBox();          ob.setObject(obj);          // Если очередь пустая - в ней еще нет элементов          if (head == null) {              // Теперь наша голова указывает на наш первый элемент              head = ob;          } else {              // Если это не первый элемент, то надо, чтобы последний элемент в очереди              // указывал на вновь прибывший элемент              tail.setNext(ob);          }          // И в любом случае нам надо наш "хвост" переместить на новый элемент          // Если это первый элемент, то "голова" и "хвост" будут указывать на один и тот же элемент          tail = ob;          // Увеличиваем размер нашей очереди          size++;      }        public Object pull() {          // Если у нас нет элементов, то возвращаем null          if (size == 0) {              return null;          }          // Получаем наш объект из вспомогательного класса из "головы"          Object obj = head.getObject();          // Перемещаем "голову" на следующий элемент          head = head.getNext();          // Если это был единственный элемент, то head станет равен null          // и тогда tail (хвост) тоже дожен указать на null.          if (head == null) {              tail = null;          }          // Уменьшаем размер очереди          size--;          // Возвращаем значение          return obj;      }        public Object get(int index) {          // Если нет элементов или индекс больше размера или индекс меньше 0          if(size == 0 || index >= size || index < 0) {              return null;          }          // Устанавлваем указатель, который будем перемещать на "голову"          ObjectBox current = head;          // В этом случае позиция равну 0          int pos = 0;          // Пока позиция не достигла нужного индекса          while(pos < index) {              // Перемещаемся на следующий элемент              current = current.getNext();              // И увеличиваем позицию              pos++;          }          // Мы дошли до нужной позиции и теперь можем вернуть элемент          Object obj = current.getObject();          return obj;      }        public int size() {          return size;      }        // Наш вспомогательный класс будет закрыт от посторонних глаз      private class ObjectBox      {          // Поле для хранения объекта          private Object object;          // Поле для указания на следующий элемент в цепочке.          // Если оно равно NULL - значит это последний элемент          private ObjectBox next;            public Object getObject() {              return object;          }            public void setObject(Object object) {              this.object = object;          }            public ObjectBox getNext() {              return next;          }            public void setNext(ObjectBox next) {              this.next = next;          }      }  } |

# Обобщения классов (Generics)

В языке Java можно описывать классы без привязки к конкретному типу данных. Например, мы собираемся записать класс Point для представления точки на плоскости с координатами x, y. И здесь перед программистом сразу встает вопрос: какой тип данных определить у этих координат? Если в будущем предполагаются только целочисленные величины x, y, то наверное, подойдет тип int. Если же будут использоваться вещественные значения, то тип float или double. Конечно, можно взять тип double и хранить с его помощью и целочисленные и вещественные значения. Но это не лучший путь, так как вещественная арифметика выполняется заметно дольше целочисленной. И для целочисленных координат лучше использовать именно целочисленный тип. Для разрешения такой ситуации в Java и была введена возможность создавать классы с обобщенными типами.

В нашем примере с классом Point его обобщенное описание можно объявить следующим образом:

**class** Point<T> {

**public** T x, y;

}

Смотрите, здесь после имени класса в угловых скобках стоит буква T – это имя обобщенного типа. На момент записи класса мы пока не знаем что будет подставлено вместо T. Здесь может быть и тип Integer и тип Double и тип String и любой другой тип языка Java (кроме примитивных: int, float, double и т.п). После этого поля x, y будут соответствовать этому типу данных. Но когда этот тип становится определенным? В момент создания экземпляра класса Point. То есть, в методе main() мы должны теперь указывать не только имя класса, но и в угловых скобках тип данных, например:

Point<Integer> pt = **new** Point<Integer>();

И, затем, использовать целочисленные координаты:

        pt.x = 10;

        pt.y = 20;

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println(pt.x + " " + pt.y);

Если потом потребуется создать объект класса Point с вещественными полями, то достаточно будет выполнить конструкцию:

Point<Double> ptD = **new** Point<Double>();

Причем ссылка ptD будет иметь уже другой тип, чем ссылка pt. И, например, вот такая строчка приведет к ошибке:

pt = ptD;   *// ошибка: разные типы ссылок*

В итоге объекты Point<Integer> и Point<Double> это, фактически, разные объекты с разными типами данных, но построенные по одному шаблону класса Point. В этом и заключается смысл обобщения.

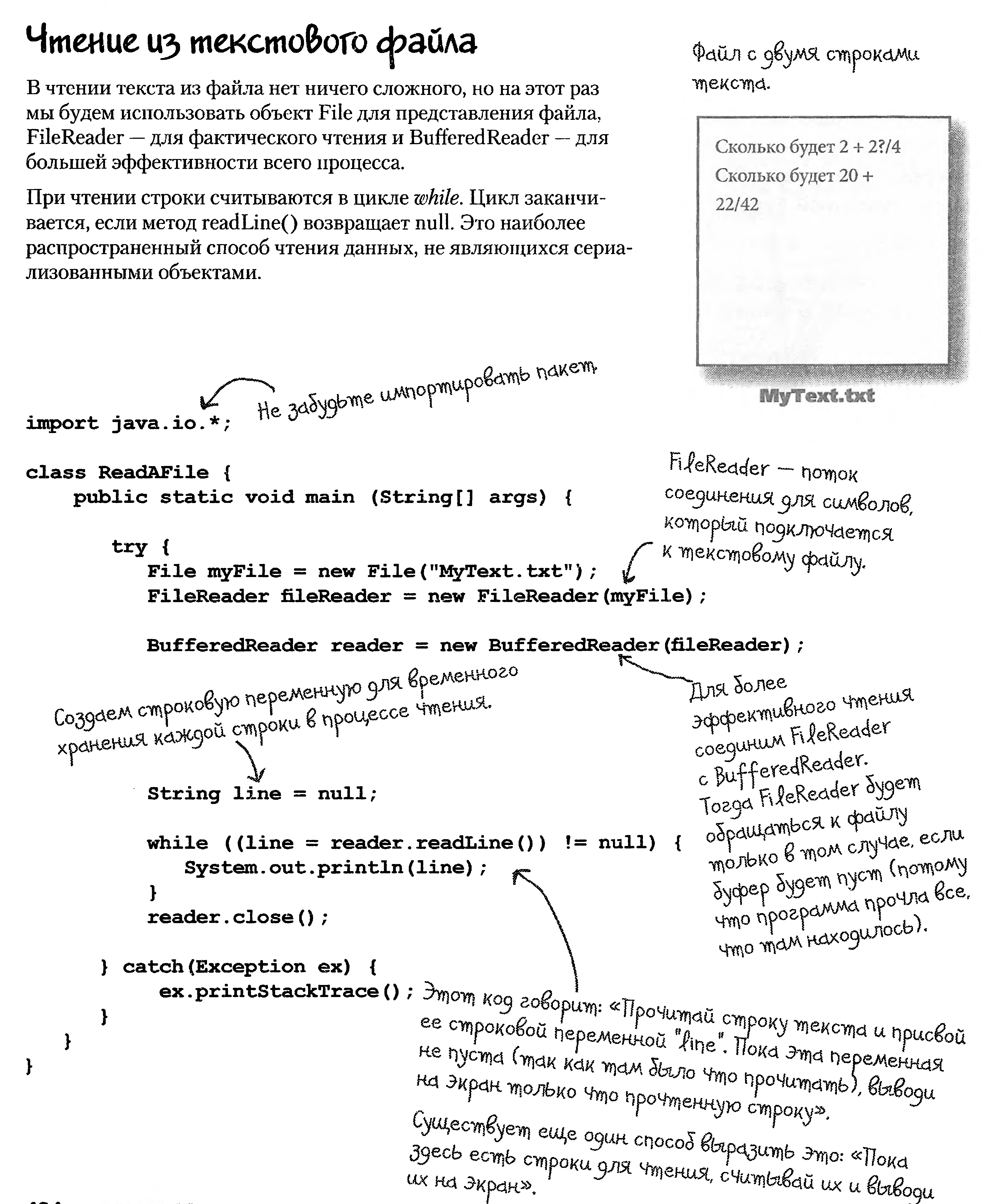
У вас может возникнуть вопрос: а почему в качестве обобщенного типа используется буква T? Можно ли вместо нее записать другую букву или слово? Да, можно. Например, вот такое объявление класса Point будет аналогично предыдущему:

**class** Point<Tt> {

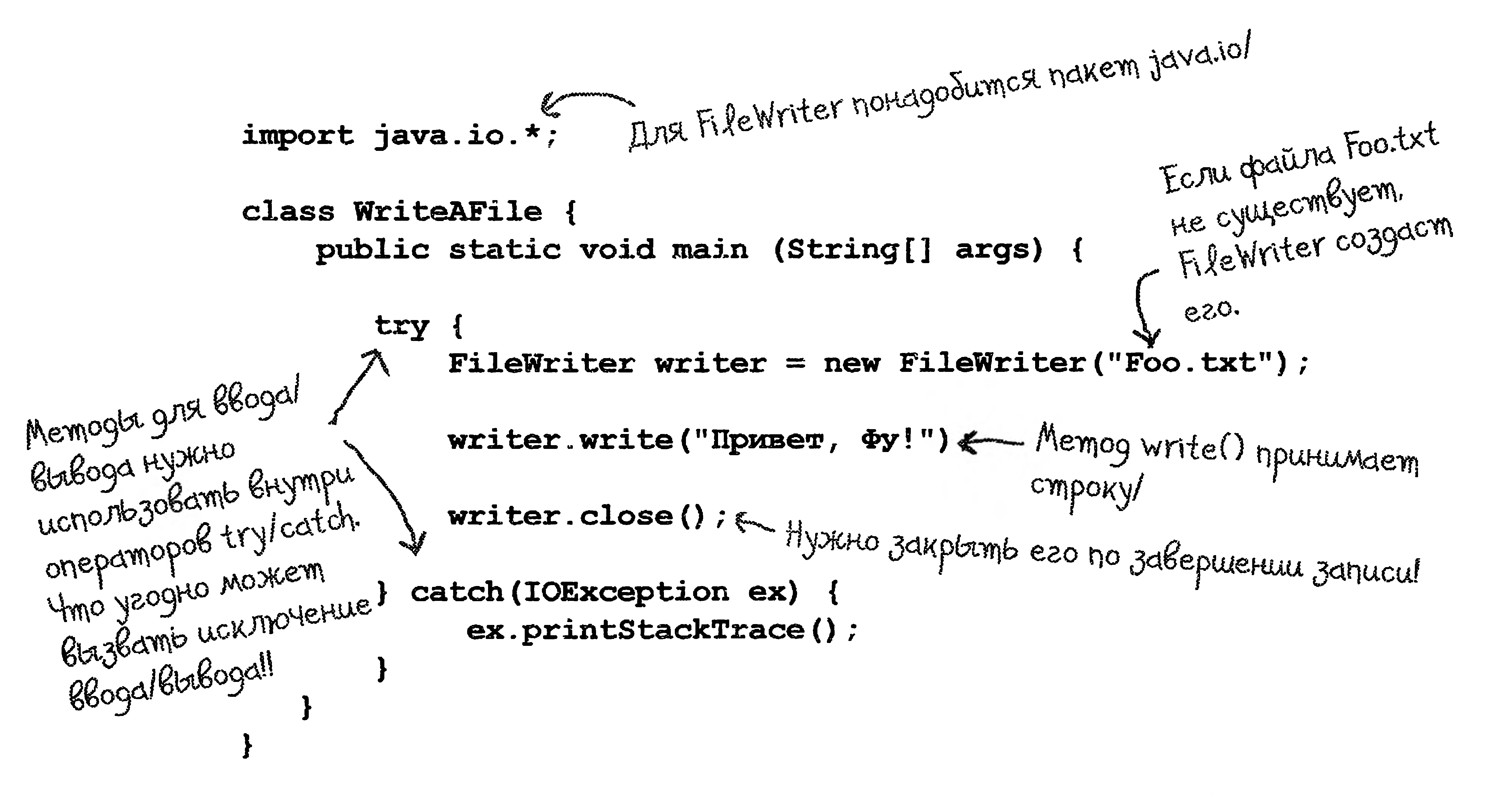
**public** Tt x, y;

}

Но, в практике программирования принято записывать просто одну букву T.







**Еще вариант записи в файл.**

