IT АКАДЕМИЯ SAMSUNG

Модуль 3. Основы программирования Android-приложений

SAMSUNG

Объектно-ориентированное проектирование

Класс в Java — это абстрактный тип данных и состоит из полей (переменных) и методов (функций).

В процессе выполнения Java-программы JVM использует определения классов для создания экземпляров классов или, как их чаще называют, объектов. В соответствии с принципом инкапуляции эти объекты содержат в себе состояние (данные) и поведение (функции).

Для того чтобы создать новую программу, необходимо произвести проектирование. Проектирование программ в объектно-ориентированной парадигме отличается от классической (например, нисходящей).

Рассмотрим примеры проектирования на разных задачах.

Объектно-ориентированное проектирование

SAMSUNG

Задача 1

Дроби. Вычисление и вывод дробей.
Прежде, чем рассмотреть процесс проектирования, рассмотрим пример. Ввести дроби A и B, вычислить и вывести на экран в виде правильной дроби. Решить без применения ООП и с применением ООП.

$$\left(\frac{An}{Ad}+3\right):\left(\frac{Bn}{Bd}-\frac{1}{3}\right)$$

Вариант 1. Решение без применения ООП

При решении задачи будем выполнять шаг за шагом последовательно, как изложено в задаче. То есть будет выполнена следующая последовательность действий.

1. Ввести значения для дробей — An, Ad, Bn, Bd.
2. Привести первую скобку к общему знаменателю и сложение числителей (An+3*Ad)/Ad.
3. Привести вторую скобку к общему знаменателю и вычитание числителей (3Bn–Bd)/3Bd.
4. Поделить результат первой операции на результат второй.
5. Вывести результат в виде числовой дроби и в виде десятичной дроби.

Объектно-ориентированное проектирование

```
Scanner sc = new Scanner(System.in);
System.out.println("Введите первую дробь");
int An = sc.nextInt();
int Ad = sc.nextInt();
System.out.println("Введите вторую дробь");
int Bn = sc.nextInt();
int Bd = sc.nextInt();
// считаем первую скобку (приводим к общему знаменателю и складываем числитель)
An = An + 3 * Ad;
// считаем вторую скобку (приводим к общему знаменателю и складываем числитель)
Bn = 3 * Bn - Bd;
Bd = 3 * Bd;
// считаем деление скобок
An = An * Bd;
Ad = Ad * Bn;
System.out.println("Результат:");
// печатаем в десятичном виде
System.out.println(1.0 * An / Ad);
if (An / Ad == 0) {
// печатаем в обычном виде
System.out.println(An);
System.out.println("---");
System.out.println(Ad);
} else {
// печатаем правильную дробь
System.out.println("   " + An % Ad);
System.out.println(An / Ad + "-----");
System.out.println("   " + Ad);
```

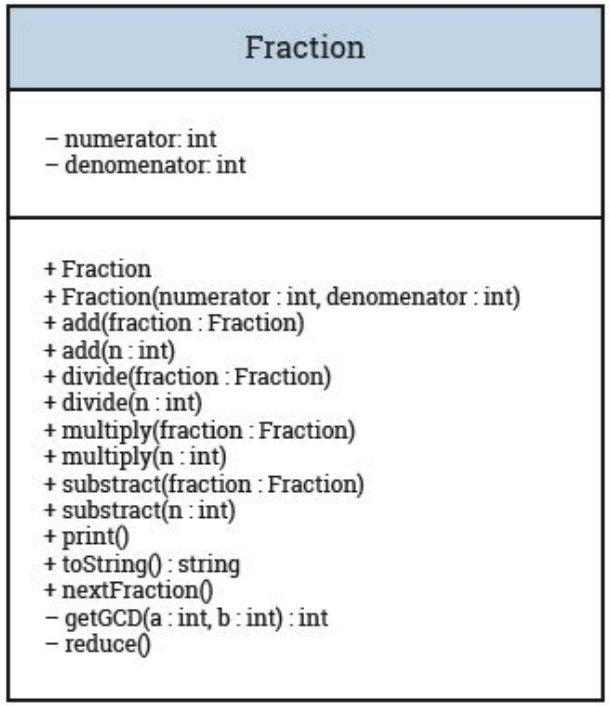
Результат работы программы:

```
Введите первую дробь
1
4
Введите вторую дробь
3
5
Результат:
12.1875
      3
12-----
     16
```

Объектно-ориентированное проектирование

Вариант 2. Решение с применением ООП

Теперь решим (FractionOOP.3.1.2) эту же задачу с применением ООП. В отличие от предыдущего примера при разработке в ООП парадигме необходимо сначала спроектировать класс так, чтобы в полях хранился числитель и знаменатель, чтобы были методы для работы с дробью — сложить, вычесть, умножить, делить, вывести на экран, чтобы был конструктор для удобного создания дроби. Кроме того, нужно перегрузить эти методы, чтобы можно было вызывать их для аргументов разных типов. И только после того, как получен этот класс, можно заниматься решением непосредственно самой задачи. Выше был словесно описан нужный класс, однако чаще используют более наглядный и компактный способ — UML диаграммы.



Объектно-ориентированное проектирование

***UML (Universal Modeling Language**, **универсальный язык моделирования)* —** это способ графического описания, применяемый для разработки программного обеспечения. Несмотря на то что есть множество разновидностей диаграмм, здесь и далее будут представлены только диаграммы классов, которые позволяют дать графическое описание классов и их связей между собой. Обычно создание диаграммы классов знаменует собой окончание процесса анализа и начало процесса проектирования.
Для построения диаграмм UML существует множество инструментов, например, ArgoUML для всех платформ, NClass для Windows, Umbrello для Linux, есть также плагины, встроенные в среды разработки.
Каждый класс в диаграмме классов UML описывается как прямоугольник, состоящий из трех выделенных блоков: наименование класса, поля класса и методы класса. При этом перед именем поля или метода указываются модификаторы видимости.

Обозначение	Модификатор Java
+	public — открытый доступ
-	private — только из методов того же класса
#	protected — только из методов этого же класса и классов, создаваемых на его основе

Объектно-ориентированное проектирование

SAMSUNG

Класс Fraction для данной диаграммы:

```
import java.util.Scanner;
public class Fraction {
    private int numerator;
    private int denominator=1;
    public void add(Fraction fraction) {
        numerator = numerator * fraction.denominator + fraction.numerator * denominator;
        denominator = denominator * fraction.denominator;
        reduce();
    }
    public void add(int n) {
        add(new Fraction(n, 1));
    }
    public void substract(Fraction fraction) {
        numerator = numerator * fraction.denominator - fraction.numerator * denominator;
        denominator = denominator * fraction.denominator;
        reduce();
    }
    public void substract(int n) {
        substract(new Fraction(n, 1));
    }
    public void multiply(Fraction fraction) {
        numerator = numerator * fraction.numerator;
        denominator = denominator * fraction.denominator;
        reduce();
    }
    public void multiply(int n) {
        multiply(new Fraction(n, 1));
    }
```

```
    public void divide(Fraction fraction) {
        if (fraction.numerator == 0) {
            System.out.println("На эту дробь делить нельзя!");
            return;
        }
        multiply(new Fraction(fraction.denominator, fraction.numerator));
    }
    public void divide(int n) {
        divide(new Fraction(n, 1));
    }
    public void nextFraction() {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int d = sc.nextInt();
        if (d == 0) {
            System.out.println("Знаменатель не может быть нулевым!");
            return;
        }
        numerator=n;
        denominator=d;
        reduce();
    }
    Fraction(){
    }
```

Объектно-ориентированное проектирование

```
Fraction(int numerator, int denominator) {
      if (denominator == 0) {
         System.out.println("Знаменатель не может быть нулевым!");
         return;
      }
      this.numerator = numerator;
      this.denominator = denominator;
      reduce();
   }
   public String toString(){
      return (numerator*denominator<0?"-":" ")+ Math.abs(numerator)+"/"+Math.abs(denominator);
   }
```

```
public void print() {
      if(numerator % denominator == 0){
         System.out.println(numerator/denominator);
         return;
      }
      if (numerator / denominator == 0) {
         System.out.println(" " + Math.abs(numerator));
         System.out.println((numerator*denominator<0?"-":" ")+" ---- или "+ 1.0 * numerator / denominator);
         System.out.println(" " + Math.abs(denominator));
      } else {
         System.out.println(" " + Math.abs(numerator % denominator));
         System.out.println((numerator*denominator<0?"-":"")+numerator / denominator + "---- или"+1.0 * numerator / denominator);
         System.out.println(" " + Math.abs(denominator));
      }
   }
   private int getGCD(int a, int b) { return b==0 ? a : getGCD(b, a%b); }
   private void reduce(){
      int t=getGCD(numerator,denominator);
      numerator/=t;
      denominator/=t;
   }
}
```

Объектно-ориентированное проектирование

Создавая по описанию класс, который содержит нужные поля и методы, программист, по сути, создает новый тип данных, который в дальнейшем можно использовать в любых программах. Поскольку класс дроби готов, то теперь можно, наконец, приступить к вычислению задачи:

```
public class Test {
    public static void main(String[] args) {
        Fraction A = new Fraction();
        Fraction B = new Fraction();
        A.nextFraction();
        B.nextFraction();
        A.add(3);
        B.substract(new Fraction(1,3));
        A.divide(B);
        A.print();
    }
}
```

Запустив на выполнение test.java и введя те же значения, получаем такой же результат, как и в прошлый раз

```
1 4
3 5
     3
12---- или 12.1875
    16
```

Объектно-ориентированное проектирование

Выводы

Задача была решена, используя два разных подхода. Сравним (при одинаковом результате).

1. В первой программе порядка 30 строк. Процесс разработки прост и понятен.
2. Во второй программе порядка 100 строк. Процесс разработки более громоздкий. Результат стал доступен только в конце.

Казалось бы, первый вариант лучше? Однако рассмотрим другой пример.

Объектно-ориентированное проектирование

Задача 2 Ввести пять правильных дробей и вычислить сумму и произведение дробей. Решить без применения ООП и с применением ООП

$$\sum_{i=0}^{5} \frac{An_i}{Ad_i} \text{ и } \prod_{i=0}^{5} \frac{An_i}{Ad_i}$$

Вариант 1. Решение без применения ООП

Решение без ООП — аналогично решению, приведенному ранее. То есть практически все предыдущее решение необходимо удалить, и решать заново, последовательно, шаг за шагом, четко следуя заданию.

1. Первоначально выполняется ввод дробей в два массива — числители в массив n, а знаменатели в массив d.
2. Для нахождения суммы в программе используется дробь rez1n/rez1d (первоначально как 0/1) и в цикле к ней прибавляются все введенные дроби.
3. Для нахождения произведения дробей в программе используется дробь rez2n/rez2d (первоначально как 1/1) и в цикле на нее умножаются все введенные дроби.
4. В конце осуществляется печать дроби rez1n/rez1d как результат суммы и rez2n/rez2d как результат произведения.

Объектно-ориентированное проектирование

```
Scanner sc = new Scanner(System.in);
System.out.println("Введите дроби:");
int n[] = new int[5];
int d[] = new int[5];
for (int i = 0; i < n.length; i++) {
    System.out.println("==============");
    n[i] = sc.nextInt();
    d[i] = sc.nextInt();
}
int rez1n = 0;
int rez1d = 1;
int rez2n = 1;
int rez2d = 1;
for (int i = 0; i < n.length; i++) {
    rez1n = rez1n * d[i] + rez1d * n[i];
    rez1d = rez1d * d[i];
    rez2n = rez2n * n[i];
    rez2d = rez2d * d[i];
}
System.out.println("Результат 1:");
// печатаем в десятичном виде
System.out.println(1.0 * rez1n / rez1d);
if (rez1n / rez1d == 0) {
    // печатаем в обычном виде
    System.out.println(rez1n);
    System.out.println("---");
    System.out.println(rez1d);
} else {
    // печатаем правильную дробь
    System.out.println("    "+rez1n%rez1d);
    System.out.println(rez1n/rez1d+"-----");
    System.out.println("    "+rez1d);
}
System.out.println("Результат 2:");
// печатаем в десятичном виде
System.out.println(1.0*rez2n/rez2d);
if (rez2n / rez2d == 0) {
    // печатаем в обычном виде
    System.out.println(rez2n);
    System.out.println("---");
    System.out.println(rez2d);
} else {
    // печатаем правильную дробь
    System.out.println("   "+rez2n%rez2d);
    System.out.println(rez2n/rez2d+"-----");
    System.out.println("   " + rez2d);
}
```

Объектно-ориентированное проектирование

Результат работы программы будет следующий:

```
Введите дроби:
==============
1
2
==============
1
2
==============
1
2
==============
1
2
==============
1
2
Результат 1:
2.5
   16
2-----
   32
Результат 2:
0.03125
1
---
32
```

Вариант 2. Решение с применением ООП

В отличие от решения этой задачи во втором варианте задачи 1 при решении с ООП на данный момент уже имеется класс дроби, и его можно просто использовать, as is, без каких бы то ни было переделок. Переписать придется только лишь ту часть, где использовались дроби, то есть метод main. Поэтому решение выглядит так (см. класс FractionOOP.3.1.2/Test1.java):

```
public class Test1 {
    public static void main(String[] args) {
        Fraction A[] = new Fraction[5];
        for (int i = 0; i < A.length; i++) {
            A[i] = new Fraction();
            A[i].nextFraction();
        }
        Fraction rez1 = new Fraction(0, 1);
        Fraction rez2 = new Fraction(1, 1);
        for (int i = 0; i < A.length; i++) {
            rez1.add(A[i]);
            rez2.multiply(A[i]);
        }
        System.out.println("Сумма");
        rez1.print();
        System.out.println("Произведение");
        rez2.print();
    }
}
```

Результат работы программы будет следующий:

```
1 2
1 2
1 2
1 2
1 2
Сумма
   1
2---- или 2.5
   2
Произведение
1
---- или 0.03125
32
```

Объектно-ориентированное проектирование

Выводы

Снова сравним решение в двух парадигмах.

1. В варианте без ООП было написано ~45 строк кода, и заново продумана реализация операций (вычисления).
2. В варианте ООП размер программы ~20 строк кода. Причем теперь уже не было нужды задумываться над тем, как именно делаются каждые операции, так как уже с прошлого решения имелась необходимая и достаточная модель дроби.

Вывод из примеров: по мере усложнения задач ООП парадигма показывает гораздо лучшие результаты в проектировании!
При разработке класса были описаны все поля и методы, которые нужны для работы с рассматриваемым объектом. Принцип инкапсуляции дает возможность скрывать часть данных и методов, то есть часть реализации класса от потребителя. Публичные же методы представляют собой интерфейс взаимодействия с этим объектом.
Степень сокрытия и тип воздействия на поля класса определяются в момент проектирования класса. Например, если разработчик не создает методы для доступа к полям класса (сеттеров), а прочие методы не меняют содержимое полей, то полученный класс называется ***immutable***. Ярким примером immutable-класса является стандартный класс String. Если в таком классе нужно выдать потребителю измененный объект (например, увеличенную дробь), то просто создается новый объект с измененным содержимым.
Чаще все же используются классы, которые позволяют менять свои данные. Для этого можно использовать методы сеттеры, либо методы, которые меняют поля по какому-то закону. В рассматриваемом случае это методы, реализующие математические операции с дробью. Наличие специальных методов для доступа (геттеров и/или сеттеров) необязательно и определяется ТЗ, однако их наличие (если это не противоречит ТЗ) скорее приветствуется, так как есть большое количество полезных фреймворков, которые используют аксессоры объектов. Например, спецификация JavaBeans требует наличие аксессоров.

Объектно-ориентированное проектирование

Задача 3

Написать игру — текстовый квест «Корпорация». Цель игрока — поднять социальный статус персонажа.

Описание

В процессе игры игроку рассказывают интерактивную историю. История начинается с первой сцены. Игрок выбирает вариант развития событий из ограниченного набора (2–3 шт.). Выбранная сцена становится текущей, и у нее также есть варианты развития событий. История заканчивается, когда у текущей ситуации нет вариантов развития событий. При выборе варианта у персонажа меняются характеристики: карьерное положение (K), активы (A), репутация P.

Объектно-ориентированное проектирование

Пример сюжета

K = 1, A = 100 тр, P = 50%

Первая сцена «Первый клиент». «Вы устроились в корпорацию менеджером по продажам программного обеспечения. Вы нашли клиента и продаете ему партию MS Windows. Ему достаточно было взять версию „HOME“ 100 коробок». Далее игроку предлагается на выбор три варианта действий.

1. Вы выпишете ему счет на 120 коробок версии «ULTIMATE» по 50 тр. Клиент может немного и переплатить, подумали вы, но за такое ведение дел вас депремировали на 10 тр, да и в глазах клиента вы показали себя не в лучшем свете. В карьере никаких изменений пока нет. K +0, A –10, P –10.
2. Вы выпишете ему счет на 100 коробок версии «PRO» по 10 тр. В принципе хорошая сделка, решили вы, и действительно вам выплатили приличный бонус, у вас намечаются положительные перспективы в карьерном росте, правда, в глазах клиента вы ничем не выделились. K +1, A +100, P +0.
3. Вы выпишете ему счет на 100 коробок версии «HOME» по 5 тр. Размер сделки поменьше, решили вы, но зато вы абсолютно честны с клиентом и продали ему только то, что ему действительно нужно, с минимальными затратами. Такое ведение бизнеса принесет вам скромный бонус, не принесет продвижение по службе, но зато в глазах клиента вы зарекомендовали себя крайне честным продавцом. K +0, A +50, P +1.

Объектно-ориентированное проектирование

SAMSUNG

Вариант 1. Решение без применения ООП

Решаем простым последовательным выполнением поставленной задачи. Для этого будем от пользователя получать номер выбранного действия и в соответствии с ним выбирать ветку дальнейшего сюжета с изменением параметров К, А, Р. Получившийся код (QuestNonOOP.3.1.3) — класс Quest.java:

```
public class Quest {

    public static void main(String[] args) {
        Scanner in  = new Scanner(System.in);
        int K=1,A=100,P=50;
        System.out.println("Вы прошли собеседование и вот-вот станете сотрудниуом Корпорации. \n Осталось уладить формальности - подпись под договором (Введите ваше имя):");
        String name;
        name=in.next();
        System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====");
        System.out.println("Только вы начали работать и тут же удача! Вы нашли клиента и продаете ему партию \n ПО МС Виндовс. Ему в принципе достаточно взять 100 коробок версии HOME.");
        System.out.println("- (1)у клиента денег много, а у меня нет - вы выпишете ему счет на 120 коробок \n ULTIMATE по 50тр");
        System.out.println("- (2)чуть дороже сделаем, кто там заметит - вы выпишете ему счет на 100 коробок \n PRO по 10тр");
        System.out.println("- (3)как надо так и сделаем - вы выпишете ему счет на 100 коробок HOME по 5тр");
        int a1=in.nextInt();
 if(a1==1){
            K+=0;A+=-10;P+=-10;
            System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====");
            // Следующие ситуации для этой ветки сюжета
        } else if(a1==2) {
            K+=1;A+=100;P+=0;
            System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====");
            // Следующие ситуации для этой ветки сюжета
        } else {
            K+=0; A+=50; P+=1;
            System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====");
            // Следующие ситуации для этой ветки сюжета
        }  System.out.println("Конец");}}
```

Объектно-ориентированное проектирование

Такое решение, даже в этом небольшом примере, выглядит громоздким. А представьте, что будет, если у вас будет по три варианта в каждом подварианте, а в нем в свою очередь еще три и так далее. Эта программа станет совершенно нечитабельной даже для создателя. Еще хуже, если вы решите впоследствии изменить сюжет, удалив или добавив несколько сцен. Тут одно неосторожное изменение может вызвать такой дисбаланс скобок, что найти потерянную скобку будет очень сложно. Конечно, можно придумать какую-то хитрую структуру данных для удобной работы с сюжетом, но зачем изобретать велосипед, если есть ООП подход?

Объектно-ориентированное проектирование

Вариант 2. Решение с применением ООП

«Объектно-ориентированный анализ и проектирование — это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами».
(Гради Буч «Объектно-ориентированный анализ и проектирование с примерами приложений на C++». Rational, Санта-Клара, Калифорния)

Прежде чем решать эту задачу в ООП парадигме, давайте разберемся, как это делать правильно. Объектно-ориентированное проектирование, как и обычное проектирование, проходит стадию анализа и синтеза. Анализ ориентирован на поиск отдельных сущностей (или классов, объектов). Синтез же воссоздаст полную модель предметной области задачи.

Анализ — это разбиение предметной области на минимальные неделимые сущности. Выделенные в процессе анализа сущности формализуются как математические модели объектов со своим внутренним состоянием и поведением.

Синтез — это соединение полученных в результате анализа моделей минимальных сущностей в единую математическую модель предметной области, с учетом взаимосвязей объектов в ней. Если же сказать простыми словами, то процесс проектирования можно описать так:

из текста подробного описания предметной области выбираем все подлежащие — это сущности (классы);
выбираем все глаголы (отглагольные формы, например, деепричастия) — это поведение сущностей (методы классов);
выбираем дополнения, определения, обстоятельства — они, скорее всего, будут определять состояние сущностей (поля классов).

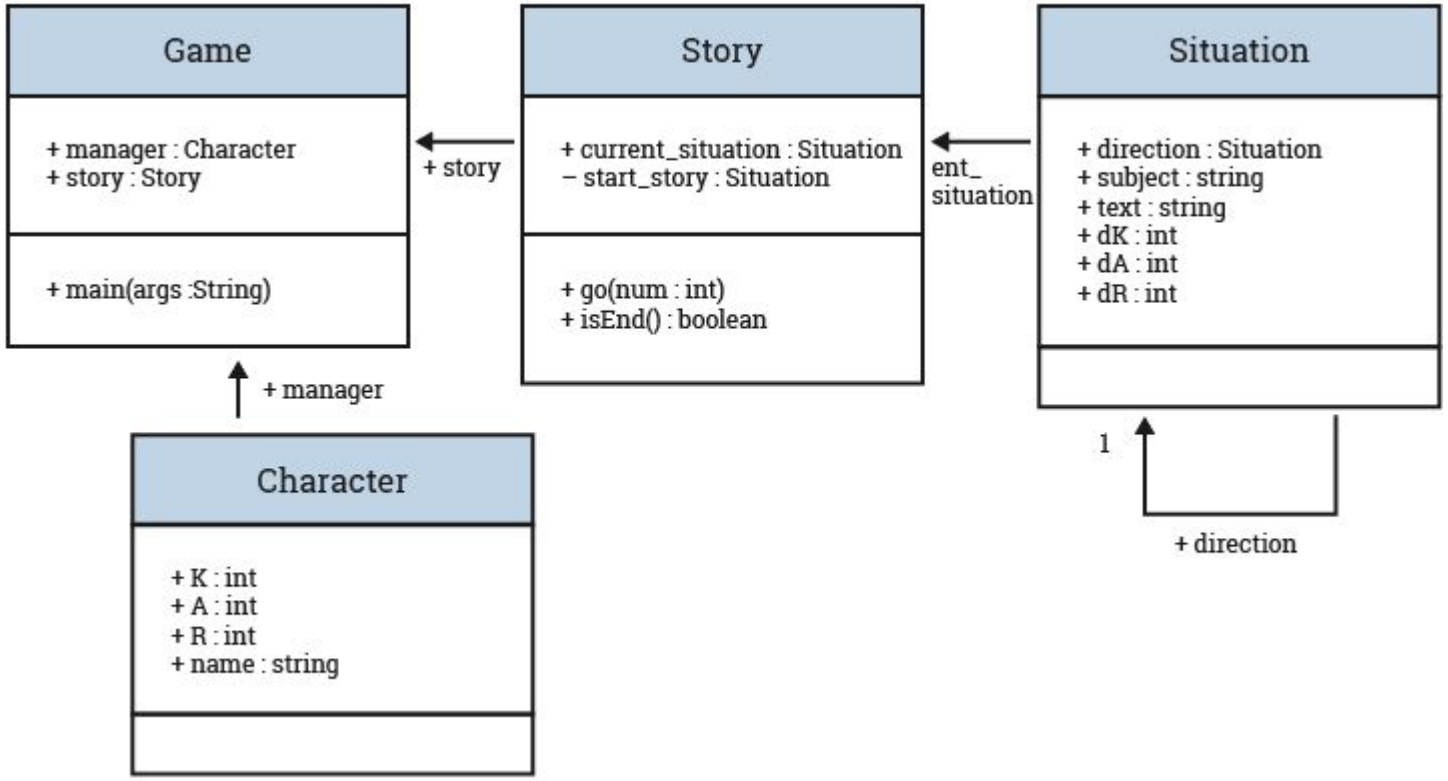
Объектно-ориентированное проектирование

Проведя синтаксический разбор и последующий анализ описания нашей задачи, получим следующую таблицу

Сущности (подлежащие)	Свойства (дополнения)	Действия (глаголы)
Игра	История, персонаж	Начать (основная программа)
Персонаж	Имя, Параметры К, А, Р	
История	Ряд сцен, текущая сцена, начальная сцена	Выбрать следующую сцену, проверить на окончание истории
Сцена	Название, описание, возможные варианты развития событий, модификаторы К, А, Р	

Объектно-ориентированное проектирование

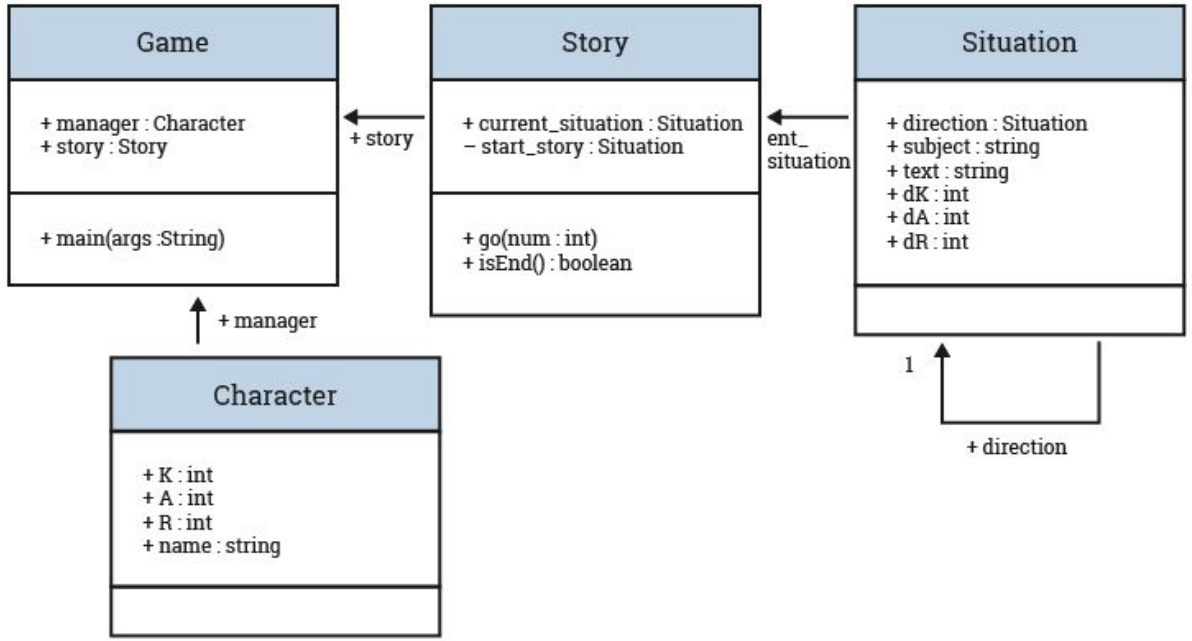
После вычленения из текста задания сущностей, их свойства и действий построим модели отдельных сущностей (это классы) и объединим их в общую модель с учетом их взаимосвязей. При описании каждой сущности следует учесть, являются ли ее поля простыми типами, или в свою очередь другими сущностями (отношение агрегирования или композиции). Это и есть разработка приложения. Сделаем это на UML и получим следующую диаграмму классов



Объектно-ориентированное проектирование

После построения модели можно прямо из UML-редактора сгенерировать исходный код (только нужно выбрать язык Java). Однако редактор зачастую дает правильный, но избыточный код. Конечно, по диаграмме можно написать классы и вручную, тем более что они небольшие.

Отдельно стоит отметить, что:
классы Character и Situation почти пусты;
класс Story в конструкторе создает всю историю как набор ситуаций;
класс Game имеет точку входа для консольного выражения (он же главный метод main), где создается персонаж и история, происходит взаимодействие с пользователем и переключение сцен.



Объектно-ориентированное проектирование

В случае же если приложение разрабатывалось для Android, то класс Game будет расширять класс Activity, и в методе onCreate (вместо main) будет создан объект персонажа и игры. Переключение сцен (метод go) должен вызываться в методах типа onClick этой активности. Кстати, если поле Direction сделать не обычным массивом, а ассоциативным, то написание самой истории (последовательности сцен) сильно упростится, так как шаги истории будут доступны не по номерам индексов в массивах вариантов, а по значащим именам сцен.

Получившийся код (QuestOOP.3.1.4) -классы Character.java, Situation.java, Story.java, Game.java:

```
//=======персонаж=======
public class Character {
    public int K;
    public int A;
    public int R;
    public String name;

    public Character(String name) {
        K = 1;
        A = 100;
        R = 50;
        this.name = name;
    }
}
```

```
//=======ситуация=======
public class Situation {
    Situation[] direction;
    String subject,text;
    int dK,dA,dR;
    public Situation (String subject, String text, int variants,
int dk,int da,int dr) {
        this.subject=subject;
        this.text=text;
        dK=dk;
        dA=da;
        dR=dr;
        direction=new Situation[variants];
    }}
```

Объектно-ориентированное проектирование

```
// =======история=======
public class Story {

    private Situation start_story;
    public Situation current_situation;

    Story() {
        start_story = new Situation(
                "первая сделка (Windows)",
                "Только вы начали работать и тут же удача! Вы нашли клиента и продаете ему "
                        + "партию ПО MS Windows. Ему в принципе достаточно взять 100 коробок версии HOME.\n"
                        + "(1)у клиента денег много, а у меня нет - вы выпишете ему счет на 120 коробок ULTIMATE по 50тр\n"
                        + "(2)чуть дороже сделаем, кто там заметит - вы выпишете ему счет на 100 коробок PRO по 10тр\n"
                        + "(3)как надо так и сделаем - вы выпишете ему счет на 100 коробок HOME по 5тр ",
                3, 0, 0, 0);
        start_story.direction[0] = new Situation(
                "корпоратив",
                "Неудачное начало, ну что ж, сегодня в конторе корпоратив! "
                        + "Познакомлюсь с коллегами, людей так сказать посмотрю, себя покажу",
                0, 0, -10, -10);
        start_story.direction[1] = new Situation(
                "совещание, босс доволен",
                "Сегодня будет совещание, меня неожиданно вызвали,"
                        + "есть сведения что \n босс доволен сегодняшней сделкой.",
                0, 1, 100, 0);
        start_story.direction[2] = new Situation(
                "мой первый лояльный клиент",
                "Мой первый клиент доволен скоростью и качеством "
                        + "моей работы. Сейчас мне звонил лично \nдиректор компании,  сообщил что скоро состоится еще более крупная сделка"
                        + " и он хотел, чтобы по ней работал именно я!", 0, 0,
                50, 1);
        current_situation = start_story;
    }

    public void go(int num) {
        if (num <= current_situation.direction.length)
            current_situation = current_situation.direction[num - 1];
        else
            System.out.println("Вы можете выбирать из "
                    + current_situation.direction.length + " вариантов");
    }

    public boolean isEnd() {
        return current_situation.direction.length == 0;
    }
}
```

Объектно-ориентированное проектирование

```
//=======игра=======
public class Game {

    public static Character manager;
    public static Story story;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Вы прошли собеседование и вот-вот станете сотрудником Корпорации. \n "
                    + "Осталось уладить формальности - подпись под договором (Введите ваше имя):");
        manager = new Character(in.next());
        story = new Story();
        while (true) {
            manager.A += story.current_situation.dA;
            manager.K += story.current_situation.dK;
            manager.R += story.current_situation.dR;
            System.out.println("=====\nКарьера:" + manager.K + "\tАктивы:"
                    + manager.A + "\tРепутация:" + manager.R + "\n=====");
            System.out.println("=============="
                    + story.current_situation.subject + "==============");
            System.out.println(story.current_situation.text);
            if (story.isEnd()) {
                System.out
                        .println("====================the end!====================");
                return;
            }
            story.go(in.nextInt());
        }
```

Объектно-ориентированное проектирование

Выводы

В предыдущем примере была создана не ООП игра-квест. Обратите внимание, что при росте количества сцен сложность восприятия кода сильно возрастает и программа быстро становится нечитабельной. В то же время при использовании ООП описание сцен довольно компактное и код получается простой и прозрачный. Можно сделать вывод об очевидном преимуществе ОО подхода при разработке ПО. Однако в этих примерах еще не был задействован весь арсенал ООП. По сути, использовалась лишь инкапсуляция. В следующей главе рассмотрим пример, в котором задействованы все преимущества ООП.

Объектно-ориентированное проектирование

Электронный журнал (мини-проект)

Необходимо разработать электронный журнал для школы. Разработку будем проводить только в ОО парадигме.

Детализируем постановку задачи. Итак, есть школа как учебное заведение, находящееся по адресу 344000, г. Ростов-на-Дону, ул. Знаний, д.1, в которой происходит обучение детей по стандартной программе среднего общего образования (11 классов). В школе работают учителя, которые преподают по несколько дисциплин, причем некоторые имеют дополнительную нагрузку в виде классного руководства либо факультативных предметов, кружков. Помимо преподавателей, в школе есть прочий персонал: директор, завуч, охранники, повара, уборщики. Вход в школу осуществляется при предъявлении магнитной карты с уникальным ID. Есть множество документов, регламентирующих процесс образования.

Объектно-ориентированное проектирование

Для реализации деятельности в этом задании выберем следующие документы:

- общий список преподавательского состава с указанием квалификации для ведения отчетности;
- общий список школьников с указанием возраста для ведения отчетности;
- общий список всех людей, имеющих доступ в школу, для выдачи магнитных карт;
- список учеников класса вместе с родителями для организации собраний;
- электронный журнал, каждая страница которого связывает отчетность о посещении/оценках школьников определенного класса по датам с учебным предметом и преподавателем.

Описывать процесс формирования журнала не будем — это общеизвестно.

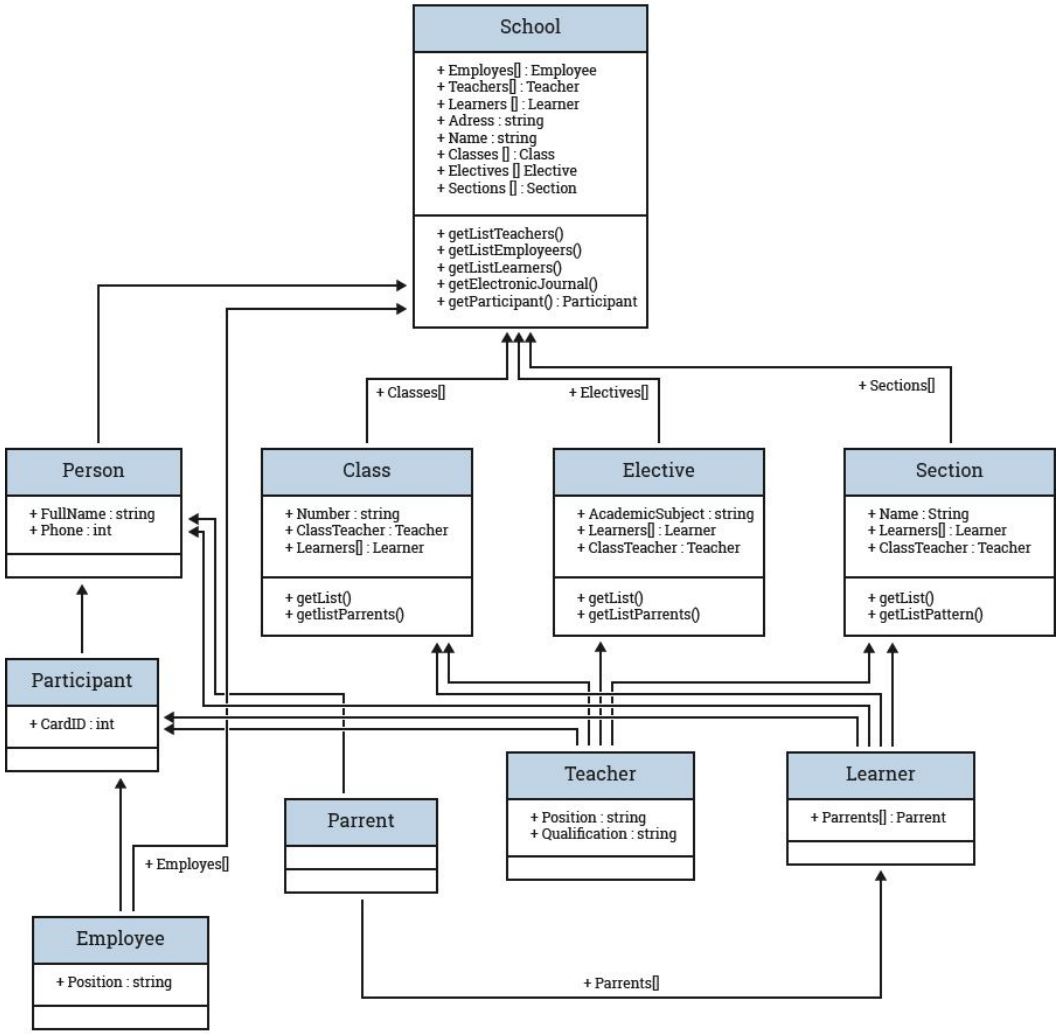
Объектно-ориентированное проектирование

Давайте проведем анализ технического задания (сущность, свойства, действия) (см. табл. 3.3).

Сущности (подлежащие)	Свойства (дополнения)	Действия (глаголы)
Школа	Сотрудники, список классов, список кружков, список факультативов	Принять на работу сотрудника, принять ученика, выдать общий список имеющих доступ в школу (CardID, ФИО), общий список учителей (ФИО, квалификация), общий список школьников (ФИО, класс, возраст), ЭлЖур
Сотрудник	CardID, ФИО, должность	
Ученик	CardID, ФИО, должность	
Учитель	CardID, ФИО, должность, квалификация	
Родитель	ФИО, номер телефона	
Класс	Номер, список учеников, классный руководитель	Выдать список учеников, список учеников с родителями
Кружок	Название, список учеников, ведущий учитель	Выдать список учеников, список учеников с родителями
Факультатив	Предмет, список учеников, ведущий учитель	Выдать список учеников, список учеников с родителями

Объектно-ориентированное проектирование

В соответствии с анализом построим UML диаграмму



Объектно-ориентированное проектирование

Итак, по описанной ранее методике была получена диаграмма классов нашей будущей программы. Однако в ней имеется ряд вопросов.

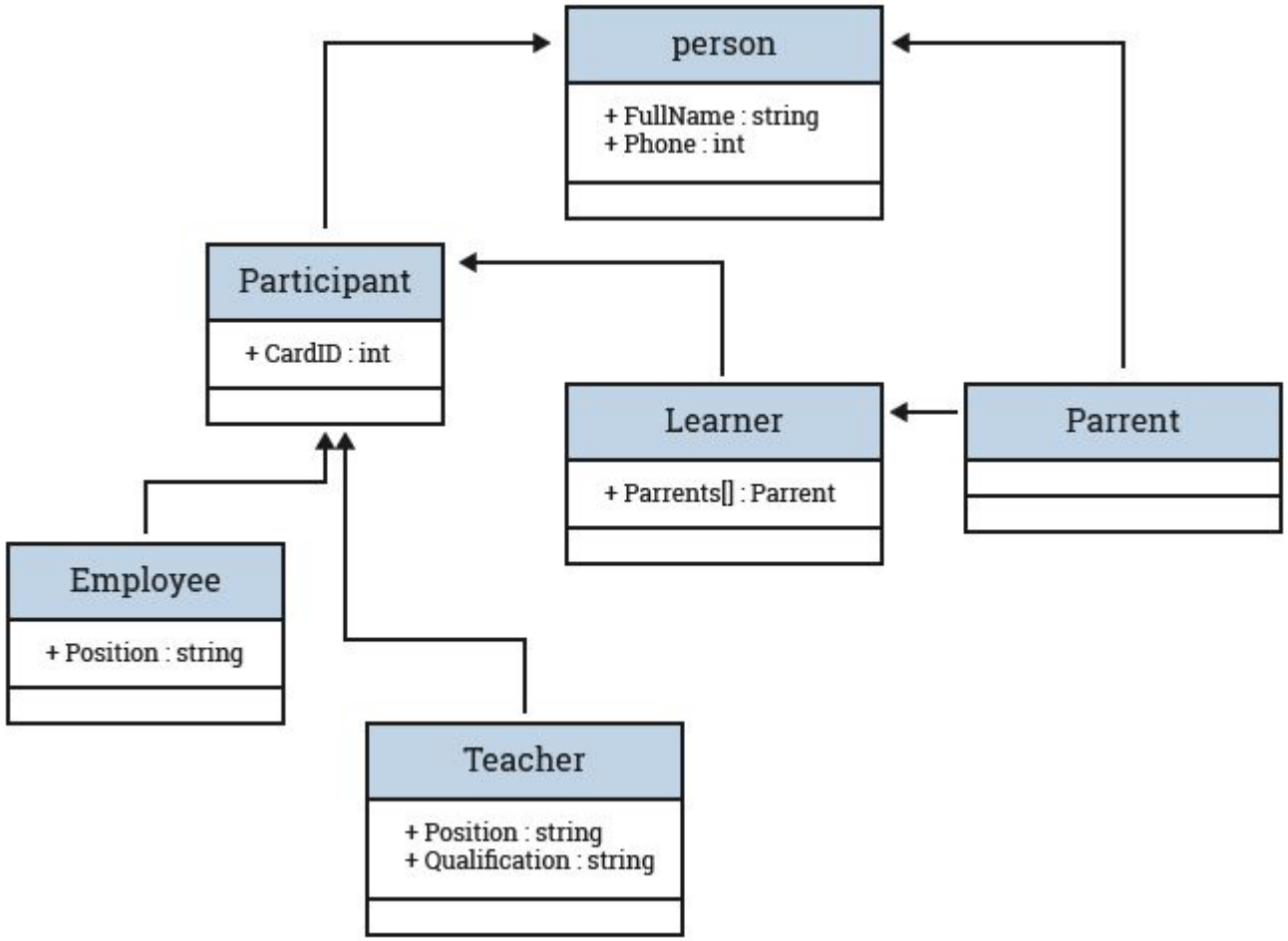
Во-первых, в некоторых классах есть повторяющиеся свойства (например, ФИО, CardID). Это наводит на мысль об общности этих классов.

Во-вторых, в постановке задачи был отчет — общий список всех людей, имеющих доступ в школу. Однако добавить такой метод в класс School не получается, так как в нашей программе нет обобщенного типа для всех участников учебного процесса, а следовательно невозможно вернуть единый массив объектов неодинаковых типов.

Для решения этих вопросов выделим суперклассы для всех участников, описанных в анализе ТЗ. Небольшой анализ показывает, что самый общий класс для всех — Person (персона, человек) с полями ФИО и телефон.

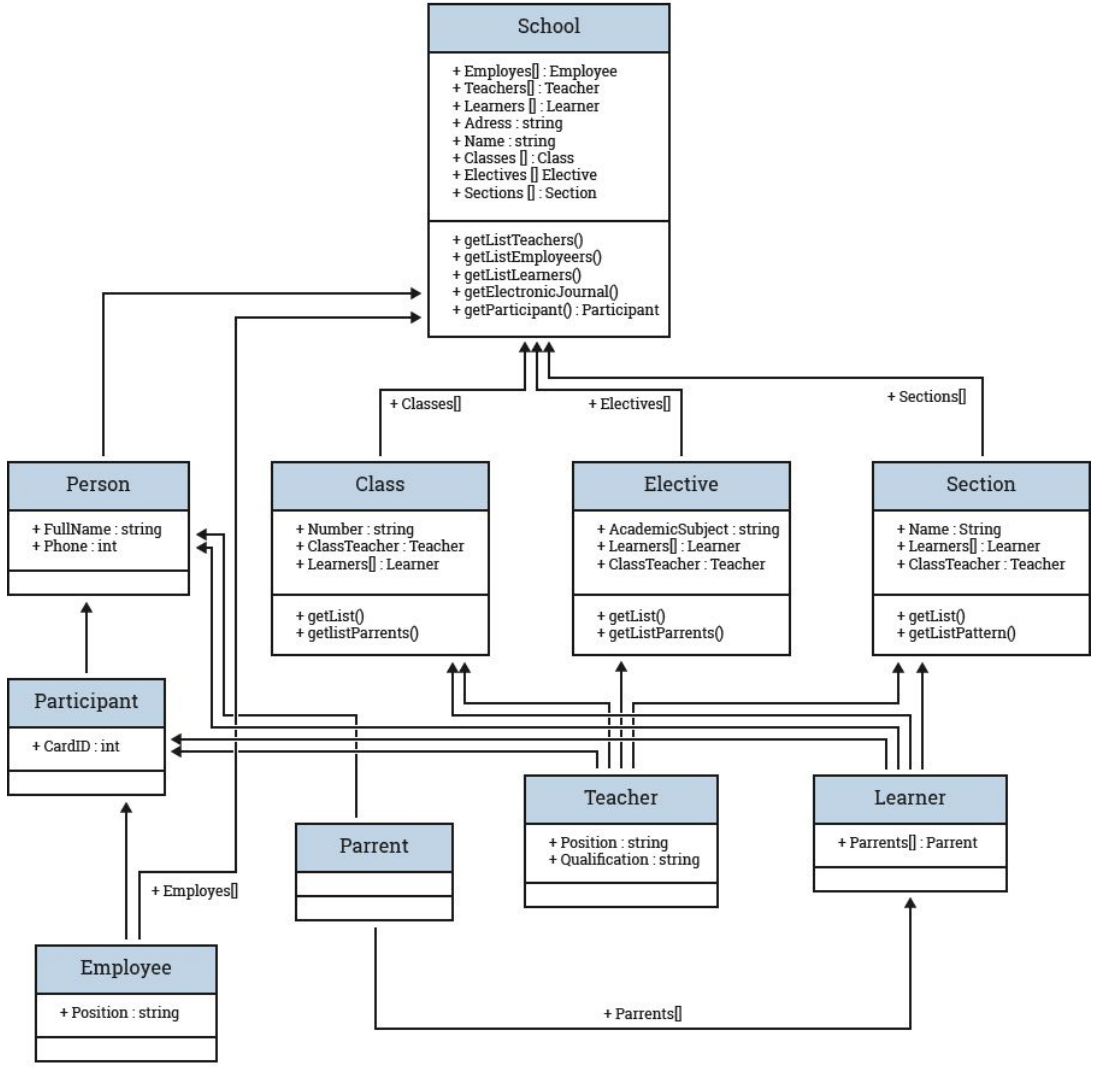
Также выделим две разновидности персон — имеющих доступ в школу (с карточкой — персонал, учителя, ученики) и не имеющих доступа (родители) (см. рис. 3.4).

Объектно-ориентированное проектирование



Объектно-ориентированное проектирование

Изменим общую модель с учетом общности сущностей (наследования классов)



В приведенной диаграмме поднятые вопросы разрешены. Теперь в классе School есть метод, который вернет массив участников учебного процесса. Обратите внимание, что вследствие полиморфизма этот метод вернет список именно участников (поля ФИО, phone, CardID) независимо от того, какие роли будут у каждого конкретного участника — ученик, учитель или сотрудник. Однако общие методы этих объектов при вызове буду вызваны именно в соответствии фактическому классу.

Конечно, в вышеприведенном UML показано только начало проектирования реальной программы, но уже видно, что использование полиморфизма и наследования дает дополнительную гибкость в проектировании.

SAMSUNG

Зачем нужна обработка исключений

Любая написанная программа будет содержать в себе проблемы, которые могут не позволить работать ей, как ожидает автор. Такие проблемы, несмотря на все их многообразие, делят на два основных класса:

- на момент написания кода имеется достаточно информации в текущем контексте, чтобы как-то справиться с трудностью. Например, ошибка синтаксиса приведет к ошибке компиляции (compile time error), которую можно легко исправить;
- исключительная ситуация — это проблема, которая мешает исполнению заданной последовательности действий в алгоритме (runtime error). Исключительная ситуация — это не ошибка в привычном понимании этого слова, она, например, может возникнуть из-за каких-то неблагоприятных внешних условий.

Ввод, вывод и исключения

В Java встроен механизм, позволяющий программисту обработать потенциально возможные исключительные ситуации, то есть написать программный код, который выполнится в случае возникновения исключительной ситуации и корректно разрешит ее.
Когда во время выполнения программа попадает в исключительную ситуацию, то у нее нет возможности продолжать обработку данных, потому что нет необходимой информации, чтобы преодолеть проблему в текущем методе (в текущем контексте). Например, программа считывает файл со счетом в игре, а кто-то удалит этот файл. Все, что можно сделать в этом случае, — это выйти из текущего контекста и отослать эту проблему к высшему контексту. Именно так и работает Java-машина (JVM) в этой ситуации. Говорят, что она «выбрасывает исключение», то есть она прерывает исполнение текущего контекста и передает управление вышестоящему контексту (функцией из которого была вызвана текущая), посылая ей дополнительную информацию о причине возникновения ситуации.

Ввод, вывод и исключения

Простой пример выбрасывания исключения — деление. Если в процессе работы программы происходит ситуация делителя на ноль, вы должны это предусмотреть. Делитель, не равный нулю, не вызовет у программы никаких затруднений, и она будет выполняться по плану, заложенному вами. Если же в переменную b будет введен 0, до перехода к блоку выполнения деления будет выброшено исключение, то есть передано обработчику исключения дальнейшее разрешение проблемы, а программа продолжит выполнение дальше.

```
// .....
int a = scanner.nextInt();
int b = scanner.nextInt();
try {
    System.err.println(a/b);
} catch (ArithmeticException e) {
    System.err.println("Произошла недопустимая
арифметическая операция");
}
// тут выполняется код после исключительной ситуации
```

Ввод, вывод и исключения

Когда выбрасывается исключение, JVM выполняет следующие действия:

- во-первых, создает объект исключения тем же способом, что и любой Java-объект — в куче (Heap Memory), при помощи new;
- затем текущий поток исполнения (который невозможно продолжать) останавливается, и ссылка на объект исключения «выталкивается» из текущего контекста. В этот момент механизм обработки исключений JVM начинает искать подходящее место для продолжения выполнения программы. Это подходящее место — обработчик исключения, чья задача — извлечь проблему, чтобы в алгоритме реакции на эту ситуацию решить проблему, или просто продолжиться, если реакция не нужна.

Также можно послать информацию об ошибке в вышестоящий контекст с помощью создания объекта исключения, представляющего вашу информацию и «выбросить» его из вашего контекста принудительно. Это называется выбрасыванием исключения. Это выглядит так:

```
if(t == null)
    throw new NullPointerException();
```

Ввод, вывод и исключения

Нередко в процессе выполнения программы могут возникать ошибки, при том необязательно по вине разработчика. Некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Так, например, может неожиданно оборваться сетевое подключение при передаче файла. Подобные ситуации называются **исключениями**.

В языке Java предусмотрены специальные средства для обработки подобных ситуаций. Одним из таких средств является конструкция **try...catch...finally**. При возникновении исключения в блоке try управление переходит в блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch.

Ввод, вывод и исключения

Например:

```
int[] numbers = new int[3];
numbers[4]=45;
System.out.println(numbers[4]);
```

Так как у нас массив numbers может содержать только 3 элемента, то при выполнении инструкции `numbers[4]=45` консоль отобразит исключение, и выполнение программы будет завершено. Теперь попробуем обработать это исключение:

```
try{
    int[] numbers = new int[3];
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){

    ex.printStackTrace();
}
System.out.println("Программа завершена");
```

При использовании блока try...catch вначале выполняются все инструкции между операторами try и catch. Если в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch. Поэтому когда выполнение программы дойдет до строки numbers[4]=45;, программа остановится и перейдет к блоку catch

Ввод, вывод и исключения

Выражение

- имеет следующий синтаксис: `catch (тип_исключения имя_переменной)`. В данном случае объявляется переменная `ex`, которая имеет тип `Exception`. Но если возникшее исключение не является исключением типа, указанного в инструкции catch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Но так как тип `Exception` является базовым классом для всех исключений, то выражение `catch (Exception ex)` будет обрабатывать практически все исключения. Обработка же исключения в данном случае сводится к выводу на консоль стека трассировки ошибки с помощью метода `printStackTrace()`, определенного в классе Exception.

После завершения выполнения блока catch программа продолжает свою работу, выполняя все остальные инструкции после блока catch.

Ввод, вывод и исключения

Конструкция try..catch также может иметь блок `finally`. Однако этот блок необязательный, и его можно при обработке исключений опускать. Блок `finally` выполняется в любом случае, возникло ли исключение в блоке try или нет:

```
try{
    int[] numbers = new int[3];
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){

    ex.printStackTrace();
}
finally{
    System.out.println("Блок finally");
}
System.out.println("Программа завершена");
```

Обработка нескольких исключений

В Java имеется множество различных типов исключений, и мы можем разграничить их обработку, включив дополнительные блоки catch:

```
int[] numbers = new int[3];
try{
    numbers[6]=45;
    numbers[6]=Integer.parseInt("gfd");
}
catch(ArrayIndexOutOfBoundsException ex){

    System.out.println("Выход за пределы массива");
}
catch(NumberFormatException ex){

    System.out.println("Ошибка преобразования из
строки в число");
}
```

Если у нас возникает исключение определенного типа, то оно переходит к соответствующему блоку catch.

Ввод, вывод и исключения

Оператор throw

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Например, в нашей программе происходит ввод числа, и мы хотим, чтобы, если число больше 30, то возникало исключение:

```
package firstapp;

import java.util.Scanner;
public class FirstApp {

    public static void main(String[] args) {

        try{
            Scanner in = new Scanner(System.in);
            int x = in.nextInt();
            if(x>=30){
                throw new Exception("Число x должно быть меньше 30");
            }
        }
        catch(Exception ex){

            System.out.println(ex.getMessage());
        }
        System.out.println("Программа завершена");
    } }
```

Здесь для создания объекта исключения используется конструктор класса Exception, в который передается сообщение об исключении. И если число x окажется больше 29, то будет выброшено исключение и управление перейдет к блоку catch.

В блоке catch мы можем получить сообщение об исключении с помощью метода `getMessage()`.

Оператор throws

Иногда метод, в котором может генерироваться исключение, сам не обрабатывает это исключение. В этом случае в объявлении метода используется оператор **throws**, который надо обработать при вызове этого метода. Например, у нас имеется метод вычисления факториала, и нам надо обработать ситуацию, если в метод передается число меньше 1:

```
public static int getFactorial(int num) throws Exception{

    if(num<1) throw new Exception("The number is less than 1");
    int result=1;
    for(int i=1; i<=num;i++){

        result*=i;
    }
    return result;
}
```

С помощью оператора `throw` по условию выбрасывается исключение. В то же время метод сам это исключение не обрабатывает с помощью try..catch, поэтому в определении метода используется выражение `throws Exception`.

Ввод, вывод и исключения

Теперь при вызове этого метода нам обязательно надо обработать выбрасываемое исключение:

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

    try{
        int result = getFactorial(-6);

        System.out.println(result);
    }
    catch(Exception ex){

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

Без обработки исключение у нас возникнет ошибка компиляции, и мы не сможем скомпилировать программу.

Ввод, вывод и исключения

В качестве альтернативы мы могли бы и не использовать оператор throws, а обработать исключение прямо в методе:

```
public static int getFactorial(int num){

    int result=1;
    try{
        if(num<1) throw new Exception("The number is less than 1");

        for(int i=1; i<=num;i++){

            result*=i;
        }
    }
    catch(Exception ex){

        System.out.println(ex.getMessage());
        result=num;
    }
    return result;
}
```

Ввод, вывод и исключения

Стратегия обработки исключения. Механизм исключений Java позволяет разработчику перехватывать потенциально некорректную работу программы, которая может возникнуть в какой-то нештатной ситуации и вызвать аварийное завершение программы либо некорректную ее работу. Однако как именно использует этот механизм разработчик, зависит только от него.

Например, худшая практика в обработчике — ничего не предпринять и просто продолжить работу. При этом пользователь не будет уведомлен о проблеме и продолжение работы программы может привести к серьезным проблемам. К примеру, вследствие ошибки файлового доступа могут быть потеряны (не сохранены) данные о текущей финансовой операции, но при подобной реакции в обработчике пользователь даже не узнает, что его финансовые данные уже неверны, хотя программа визуально продолжит работать нормально.

Правильнее, даже если невозможно исправить ситуацию, просто уведомить пользователя о проблеме и завершить работу программы (метода). Такая модель обработки исключительных ситуаций называется ***прерыванием***.

Ввод, вывод и исключения

Альтернативная модель называется возобновлением. При такой модели предполагается, что в обработчике предпринимаются меры по предотвращению причины нештатной ситуации, после чего начинается повторная попытка выполнения критичного блока. Например, если при считывании данных из сети происходит исключительная ситуацию, связанная с обрывом соединения, в обработчике можно попытаться возобновить соединение (либо подключиться к альтернативному серверу), после чего попытаться опять считывать данные. Для реализации такой модели можно использовать рекурсию или поместить блок try-catch в цикл while:

```
while(true) {
    try {
        //код, выбрасывающий исключение
        break;
    }catch(IOException e) {
        // Здесь обрабатывается исключение для
восстановления
    }
}
```

Ввод, вывод и исключения

Обработчик «заглушка». Можно создать обработчик, «ловящий» все типы исключений. Для этого в блоке catch необходимо перехватить исключение базового типа Exception:

```
catch(Exception e) {
    System.err.println("Что то пошло не так!");
}
```

Этот код поймает любое исключение, поэтому его нужно помещать в конце списка обработчиков для предотвращения перехвата у любого более специализированного обработчика.

Классы исключений

Базовым классом для всех исключений является класс **Throwable**. От него уже наследуются два класса: **Error** и **Exception**. Все остальные классы являются производными от этих двух классов.
Класс Error описывает внутренние ошибки в исполняющей среде Java. Программист имеет очень ограниченные возможности для обработки подобных ошибок.
Собственно исключения наследуются от класса Exception. Среди этих исключений следует выделить класс **RuntimeException**. RuntimeException является базовым классом для так называемой группы **непроверяемых исключений** (unchecked exceptions) - компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором throws в объявлении метода. Такие исключения являются следствием ошибок разработчика, например, неверное преобразование типов или выход за пределы массива.

Ввод, вывод и исключения

Некоторые из классов непроверяемых исключений:

- **ArithmeticException**: исключение, возникающее при делении на ноль
- **IndexOutOfBoundException**: индекс вне границ массива
- **IllegalArgumentException**: использование неверного аргумента при вызове метода
- **NullPointerException**: использование пустой ссылки
- **NumberFormatException**: ошибка преобразования строки в число

Все остальные классы, образованные от класса Exception, называются проверяемыми исключениями (checked exceptions).

Некоторые из классов проверяемых исключений:

- **CloneNotSupportedException**: класс, для объекта которого вызывается клонирование, не реализует интерфейс Cloneable
- **InterruptedException**: поток прерван другим потоком
- **ClassNotFoundException**: невозможно найти класс

Ввод, вывод и исключения

Подобные исключения обрабатываются с помощью конструкции try..catch. Либо можно передать обработку методу, который будет вызывать данный метод, указав исключения после оператора throws:

```
public Person clone() throws
CloneNotSupportedException{

    Person p = (Person) super.clone();
    return p;
}
```

Ввод, вывод и исключения

В итоге получается следующая иерархия исключений:

Например:

```
try{
    int x = 6/0;
}
catch(Exception ex){

    ex.printStackTrace();
}
```

Поскольку все классы исключений наследуются от класса Exception, то все они наследуют ряд его методов, которые позволяют получить информацию о характере исключения. Среди этих методов отметим наиболее важные:

Метод getMessage() возвращает сообщение об исключении

Метод getStackTrace() возвращает массив, содержащий трассировку стека исключения

Метод printStackTrace() отображает трассировку стека

Ввод, вывод и исключения

Функция	Описание
String getMessage( )	Возвращает сообщение об исключении
String toString( )	Возвращает короткое описание исключения
void printStackTrace( ), void printStackTrace(PrintStream), void printStackTrace(PrintWriter)	Выдача в стандартный или указанный поток полной информации о точке возникновения исключения

Кроме того, в базовом классе Throwable Object (базовый тип для всего) существуют другие методы. Один из них, который может быть удобен для исключений, это getClass(), который возвращает объектное представление класса этого объекта. Вы можете спросить у объекта этого класса его имя с помощью getName() или toString(). Обычно эти методы не используются при обработке исключений.

Ввод, вывод и исключения

```
public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Пробное исключение");
        } catch(Exception e) {
            System.err.println("Обрабатываем исключение");
            System.err.println("e.getMessage(): " + e.getMessage());
            System.err.println("e.getLocalizedMessage(): "
+e.getLocalizedMessage());
            System.err.println("e.toString(): " + e);
            System.err.println("e.printStackTrace():");
            e.printStackTrace(System.err);
        }
    }
}
```

Вывод этой программы:

```
Обрабатываем исключение
e.getMessage(): Пробное исключение
e.getLocalizedMessage(): Пробное исключение
e.toString(): java.lang.Exception: Пробное исключение
e.printStackTrace():
java.lang.Exception: Пробное исключение
        at
ExceptionMethods.main(ExceptionMethods.java:7)
```

В этом примере методы обеспечивают больше информации, чем просто тип исключения. Таким образом, имея на вооружении данные методы, мы всегда можем точно определить, где и почему произошло исключение, что может помочь не столько в обработке исключений, сколько в отладке программы.

Класс File. Работа с файлами и каталогами

Класс File, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом File.
В зависимости от того, что должен представлять объект File - файл или каталог, мы можем использовать один из конструкторов для создания объекта:

```
File(String путь_к_каталогу)
File(String путь_к_каталогу, String имя_файла)
File(File каталог, String имя_файла)
```

Например:

```
// создаем объект File для каталога
File dir1 = new File("C://SomeDir");
// создаем объекты для файлов, которые находятся в каталоге
File file1 = new File("C://SomeDir", "Hello.txt");
File file2 = new File(dir1, "Hello2.txt");
```

Ввод, вывод и исключения

Класс File имеет ряд методов, которые позволяют управлять файлами и каталогами. Рассмотрим некоторые из них:

- **boolean createNewFile()**: создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает true, иначе false
- **boolean delete()**: удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает true.
- **boolean exists()**: проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает true, иначе возвращает false
- **String getAbsolutePath()**: возвращает абсолютный путь для пути, переданного в конструктор объекта
- **String getName()**: возвращает краткое имя файла или каталога
- **String getParent()**: возвращает имя родительского каталога
- **boolean isDirectory()**: возвращает значение true, если по указанному пути располагается каталог
- **boolean isFile()**: возвращает значение true, если по указанному пути находится файл
- **boolean isHidden()**: возвращает значение true, если каталог или файл являются скрытыми
- **long length()**: возвращает размер файла в байтах
- **long lastModified()**: возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи Unix
- **String[] list()**: возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге
- **File[] listFiles()**: возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге
- **boolean mkdir()**: создает новый каталог и при удачном создании возвращает значение true
- **boolean renameTo(File dest)**: переименовывает файл или каталог

Работа с каталогами

Если объект File представляет каталог, то его метод `isDirectory()` возвращает `true`. И поэтому мы можем получить его содержимое - вложенные подкаталоги и файлы с помощью методов `list()` и `listFiles()`. Получим все подкаталоги и файлы в определенном каталоге:

```
import java.io.File;

public class Program {

    public static void main(String[] args) {
        // определяем объект для каталога
        File dir = new File("C://SomeDir");
        // если объект представляет каталог
        if(dir.isDirectory())
        {
            // получаем все вложенные объекты в каталоге
            for(File item : dir.listFiles()){
                if(item.isDirectory()){
                    System.out.println(item.getName() + "  \t folder");
                }
                else{
                    System.out.println(item.getName() + "\t file");
                }
             }
        }
    }
}
```

Ввод, вывод и исключения

Теперь выполним еще ряд операций с каталогами, как удаление, переименование и создание:

```
import java.io.File;

public class Program {

    public static void main(String[] args) {

        // определяем объект для каталога
        File dir = new File("C://SomeDir//NewDir");
        boolean created = dir.mkdir();
        if(created)
            System.out.println("Folder has been created");
        // переименуем каталог
        File newDir = new File("C://SomeDir//NewDirRenamed");
        dir.renameTo(newDir);
        // удалим каталог
        boolean deleted = newDir.delete();
        if(deleted)
            System.out.println("Folder has been deleted");
    }
}
```

Работа с файлами

Работа с файлами аналогична работе с каталога. Например, получим данные по одному из файлов и создадим еще один файл:

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

public class Program {

    public static void main(String[] args) {

        // определяем объект для каталога
        File myFile = new File("C://SomeDir//notes.txt");
        System.out.println("File name: " + myFile.getName());
        System.out.println("Parent folder: " + myFile.getParent());
        if(myFile.exists())
            System.out.println("File exists");
        else
            System.out.println("File not found");

        System.out.println("File size: " + myFile.length());
        if(myFile.canRead())
            System.out.println("File can be read");
        else
            System.out.println("File can not be read");
        if(myFile.canWrite())
            System.out.println("File can be written");
        else
            System.out.println("File can not be written");

        // создадим новый файл
        File newFile = new File("C://SomeDir//MyFile");
        try
        {
            boolean created = newFile.createNewFile();
            if(created)
                System.out.println("File has been created");
        }
        catch(IOException ex){

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

Чтение и запись файлов. FileInputStream и FileOutputStream

Запись файлов и класс FileOutputStream

Класс **FileOutputStream** предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность.
Через конструктор класса FileOutputStream задается файл, в который производится запись. Класс поддерживает несколько конструкторов:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)
```

Файл задается либо через строковый путь, либо через объект File. Второй параметр - append задает способ записи: если он равен true, то данные дозаписываются в конец файла, а при false - файл полностью перезаписывается

Ввод, вывод и исключения

Например, запишем в файл строку:

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        String text = "Hello world!"; // строка для записи
        try(FileOutputStream fos=new
FileOutputStream("C://SomeDir//notes.txt"))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();

            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
        System.out.println("The file has been written");
    }
}
```

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Если такого файла нет, то он автоматически создается при записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода write строка записывается в файл.

Для автоматического закрытия файла и освобождения ресурса объект FileOutputStream создается с помощью конструктции try...catch.

При этом необязательно записывать весь массив байтов. Используя перегрузку метода `write()`, можно записать и одиночный байт:

```
fos.write(buffer[0]); // запись первого байта
```

Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс **FileInputStream**, который является наследником класса InputStream и поэтому реализует все его методы.
Для создания объекта FileInputStream мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

```
FileInputStream(String fileName) throws FileNotFoundException
```

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение **FileNotFoundException**.

Ввод, вывод и исключения

Считаем данные из ранее записанного файла и выведем на консоль:

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("C://SomeDir//notes.txt"))
        {
            System.out.printf("File size: %d bytes \n", fin.available());

            int i=-1;
            while((i=fin.read())!=-1){

                System.out.print((char)i);
            }
        }
        catch(IOException ex){

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

В данном случае мы считываем каждый отдельный байт в переменную i:

```
while((i=fin.read())!=-1){
```

Когда в потоке больше нет данных для чтения, метод возвращает число -1.

Затем каждый считанный байт конвертируется в объект типа char и выводится на консоль.

Ввод, вывод и исключения

Подобным образом можно считать данные в массив байтов и затем производить с ним манипуляции:

```
byte[] buffer = new byte[fin.available()];
// считаем файл в буфер
fin.read(buffer, 0, fin.available());

System.out.println("File data:");
for(int i=0; i<buffer.length;i++){

    System.out.print((char)buffer[i]);
}
```

Ввод, вывод и исключения

Совместим оба класса и выполним чтение из одного и запись в другой файл:

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("C://SomeDir//notes.txt");
                FileOutputStream fos=new FileOutputStream("C://SomeDir//notes_new.txt"))
        {
            byte[] buffer = new byte[fin.available()];
            // считываем буфер
            fin.read(buffer, 0, buffer.length);
            // записываем из буфера в файл
            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

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

Классы FileInputStream и FileOutputStream предназначены прежде всего для записи двоичных файлов, то есть для записи и чтения байтов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

Класс PrintStream

Класс PrintStream - это именно тот класс, который используется для вывода на консоль. Когда мы выводим на консоль некоторую информацию с помощью вызова `System.out.println()`, то тем самым мы задействует **PrintStream**, так как переменная `out` в классе System как раз и представляет объект класса PrintStream, а метод `println()` - это метод класса PrintStream.

Но PrintStream полезен не только для вывода на консоль. Мы можем использовать данный класс для записи информации в поток вывода. Для этого PrintStream определяет ряд конструкторов:

```
PrintStream(OutputStream outputStream)
PrintStream(OutputStream outputStream, boolean autoFlushingOn)
PrintStream(OutputStream outputStream, boolean autoFlushingOn, String charSet) throws UnsupportedEncodingException
PrintStream(File outputFile) throws FileNotFoundException
PrintStream(File outputFile, String charSet) throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String outputFileName) throws FileNotFoundException
PrintStream(String outputFileName, String charSet) throws FileNotFoundException, UnsupportedEncodingException
```

Параметр outputStream - это объект OutputStream, в который производится запись. Параметр autoFlushingOn при значении true позволяет автоматически записывать данные в поток вывода. По умолчанию этот параметр равен false. Параметр charSet позволяет указать кодировку символов.

В качестве источника для записи данных вместо OutputStream можно использовать объект File или строковый путь, по которому будет создаваться файл.

Ввод, вывод и исключения

Для вывода информации в выходной поток PrintStream использует следующие методы:
println(): вывод строковой информации с переводом строки
print(): вывод строковой информации без перевода строки
printf(): форматированный вывод

Например, запишем информацию в файл:

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        String text = "Привет мир!"; // строка для записи
        try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes3.txt");
                PrintStream printStream = new PrintStream(fos))
        {
            printStream.println(text);
            System.out.println("Запись в файл произведена");
        }
        catch(IOException ex){

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

В данном случае применяется форма конструктора PrintStream, которая в качестве параметра принимает поток вывода: PrintStream (OutputStream out). Кроме того, мы могли бы использовать ряд других форм конструктора, например, указывая названия файла для записи: PrintStream (string filename)

В качестве потока вывода используется объект FileOutputStream. С помощью метода println() производится запись информации в выходной поток - то есть в объект FileOutputStream. (В случае с выводом на консоль с помощью System.out.println() в качестве потока вывода выступает консоль)

Ввод, вывод и исключения

Кроме того, как и любой поток вывода и наследник класса OutputStream он имеет метод write:

```
import java.io.*;
public class Program {

    public static void main(String[] args) {

        try(PrintStream printStream = new PrintStream("notes3.txt"))
        {
            printStream.print("Hello World!");
            printStream.println("Welcome to Java!");

            printStream.printf("Name: %s Age: %d \n", "Tom", 34);

            String message = "PrintStream";
            byte[] message_toBytes = message.getBytes();
            printStream.write(message_toBytes);

            System.out.println("The file has been written");
        }
        catch(IOException ex){

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

После выполнения этой программы получится файл со следующим содержанием:

Hello World!Welcome to Java!
Name: Tom Age: 34
PrintStream

SAMSUNG

PrintWriter

На PrintStream похож другой класс PrintWriter. Его можно использовать как для вывода информации на консоль, так и в файл или любой другой поток вывода. Данный класс имеет ряд конструкторов:

PrintWriter(File file): автоматически добавляет информацию в указанный файл
PrintWriter(File file, String csn): автоматически добавляет информацию в указанный файл с учетом кодировки csn
PrintWriter(OutputStream out): для вывода информации используется существующий объект OutputStream, автоматически сбрасывая в него данные
PrintWriter(OutputStream out, boolean autoFlush): для вывода информации используется существующий объект OutputStream, второй параметр указывает, надо ли автоматически добавлять в OutputStream данные
PrintWriter(String fileName): автоматически добавляет информацию в файл по указанному имени
PrintWriter(String fileName, String csn): автоматически добавляет информацию в файл по указанному имени, используя кодировку csn
PrintWriter(Writer out): для вывода информации используется существующий объект Writer, в который автоматически идет запись данных
PrintWriter(Writer out, boolean autoFlush): для вывода информации используется существующий объект Writer, второй параметр указывает, надо ли автоматически добавлять в Writer данные

Ввод, вывод и исключения

PrintWriter реализует интерфейсы Appendable, Closable и Flushable, и поэтому после использования представляемый им поток надо закрывать.

Для записи данных в поток он также используется методы printf() и println().

Например, применим данный класс для вывода на консоль:

```
try(PrintWriter pw = new PrintWriter(System.out))
{
    pw.println("Hello world!");
}
```

В качестве потока вывода здесь применяется System.out, а на консоль будет выведена строка "Hello world!"

Ввод, вывод и исключения

Класс Scanner

Начиная с версии 1.5, в Java появился класс Scanner в пакете java.util. При создании объекта ему в качестве аргумента могут передаваться файл или поток для считывания. Далее для каждого из базовых типов <T> имеется пара методов:
hasNext() проверяет можно ли далее прочесть данные типа T;
next() для считывания данных этого типа.
Например, метод nextInt() считывает очередной int, а метод hasNextDouble() возвращает истину или ложь в зависимости от того, есть ли в потоке следующее значение double для чтения. Например, чтение строк файла a.txt и вывод их в консоль построчно:

```
try {
    File file = new File("d:/a.txt");
    Scanner scanner = new Scanner(file);
    while (scanner.hasNext()) {
        System.out.println(scanner.next());
    }
    scanner.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Хотя Scanner и не является потоком, у него тоже необходимо вызывать метод close(), который закроет используемый за основной источник поток.

Несмотря на то что обычно при завершении программы на Java, открытые файлы закрываются автоматически, все же рекомендуется обеспечить корректное закрытие потоков даже при возможном возникновении исключений. Для этого удобнее всего метод close() включать в блок finally.

Ввод, вывод и исключения

Пример 3.2

IT School SAMSUNG – программа дополнительного образования по основам IT и программирования. Она создана компанией Samsung с целью обучить 5 000 школьников в более чем 20 городах России в течение 5 лет.

То выходной файл *out.txt*:

IT Школа SAMSUNG – программа дополнительного образования по основам IT и программирования. Она создана компанией Samsung с целью обучить 5 000 школьников в более чем 20 городах России в течение 5 лет.

Для решения поставленной задачи в создаваемом методе, назовем его replace, воспользуемся низкоуровневыми инструментами Scanner и PriontWriter:

```
void replace(String fileIn,String fileOut) {
    File in=new File(fileIn);
    File out=new File(fileOut);
    Scanner sc=new Scanner(in);
    PrintWriter pw=new PrintWriter(out);
        // код копирования будет здесь
}
```

Ввод, вывод и исключения

Далее в цикле будем пословно считывать исходный файл и писать в целевой, с заменой слова, конечно. Ниже приведен код копирования:

```
while(sc.hasNext()){
    String word=sc.next();
    if(word.equals("School"))
            word="Школа";
    pw.print(word+" ");
}
```

Важно не забыть закрыть файлы в конце:

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

Ввод, вывод и исключения

При работе с файлами часто имеются критические блоки кода, требующие обработки исключений. В имеющейся постановке задачи недостаточно данных для качественной обработки, поэтому принимаем решение делегировать обработку вышестоящему контексту, добавив в описание функции ключевого слова throws и типа делегируемого исключения. В итоге функция имеет следующий вид:

```
void replace(String fileIn,String fileOut) throws FileNotFoundException{
    File in=new File(fileIn);
    File out=new File(fileOut);
    Scanner sc=new Scanner(in);
    PrintWriter pw=new PrintWriter(out);
    while(sc.hasNext()){
        String word=sc.next();
        if(word.equals("School"))
                word="Школа";
        pw.print(word+" ");
    }
    sc.close();
    pw.close();
}
```

Ввод, вывод и исключения

Протестировать использование функции replace можно следующим образом:

```
public static void main(String[] args) {
    FileIO test=new FileIO();
    try{
     test.replace("in.txt", "out.txt");
    }catch(Exception ex){
        System.out.println("Что то пошло не так: "+ex.getMessage());
    }
}
```