Модуль 2. Объектно-ориентированное программирование

Тема 3.2. Ввод, вывод и исключения

2 часа

Оглавление

Тема 3.2. Ввод, вывод и исключения

3.2.1. Зачем нужна обработка исключений

Исключительная ситуация - это проблема, которая мешает последовательному исполнению метода или ограниченного участка программы. Важно различать исключительные состояния и обычные проблемы, в которых имеется достаточно информации в текущем контексте, чтобы как-то справиться с трудностью. Например, в программе не закрыты фигурные скобки и это приведет к ошибке компиляции, которую можно легко исправить.

В исключительном состоянии нет возможности продолжать обработку данных, потому что нет необходимой информации, чтобы разобраться с проблемой в текущем контексте. Например, программа считывает файл со счётом 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 это происходит, когда **выбрасывается исключение**.

Простой пример выбрасывания исключения - деление. Если в процессе работы программы происходит ситуация обнуления делителя, вы должны это предусмотреть. Делитель, не равный нулю, не должен вызвать у программы никаких затруднений и она будет выполняться по плану, заложенному вами. Если же произойдет ситуация, кода делитель будет равен нулю, необходимо **до** перехода к блоку выполнения деления выбросить исключение, т.е. передать **обработчику исключения** дальнейшее разрешение проблемы, а программа продолжит выполнение дальше.

```
int a = 4;
try {
   System.err.println(a/0);
} catch (ArithmeticException e) {
    System.err.println("Произошла недопустимая арифметическая операция");
}
```

Когда **выбрасывается исключение**, происходит несколько вещей.

- Во-первых, создается объект исключения тем же способом, что и любой Java объект: в куче, с помощью *new*.
- Затем текущий путь выполнения (который невозможно продолжать) останавливается, и ссылка на объект исключения выталкивается из текущего контекста. В этот момент вступает механизм обработки исключений и начинает искать подходящее место для продолжения выполнения программы. Это подходящее место - **обработчик исключения**, чья работа - извлечь проблему, чтобы программа могла попробовать другой способ, либо просто продолжиться.

Вы можете послать информацию об ошибке в больший контекст с помощью создания объекта, представляющего вашу информацию и “выбросить” его из вашего контекста. Это называется выбрасыванием исключения. Это выглядит так:

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

Здесь выбрасывается исключение, которое позволяет в текущем контексте отказаться от ответственности, думая о будущем решении.

3.2.2. Ключевое слово throw

Часть исключений может обрабатывать сама система. Но можно создать собственные исключения при помощи оператора *throw*. Код выглядит так:

```
throw экземпляр_Throwable
```

Вам нужно создать экземпляр класса *Throwable* или его наследников. Получить объект класса Throwable можно в операторе *catch* или стандартным способом через оператор *new*.

Поток выполнения останавливается непосредственно после оператора *throw* и другие операторы не выполняются. При этом ищется ближайший блок *try/catch,* соответствующего исключению типа.

```
try {
        throw new NullPointerException("Объекта не существует");
   } catch (NullPointerException e) {
        Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
   }
```

В этом примере создан новый объект класса *NullPointerException*. Многие классы исключений кроме стандартного конструктора по умолчанию с пустыми скобками имеют второй конструктор с строковым параметром, в котором можно разместить подходящую информацию об исключении. Получить текст из него можно через метод *getMessage()*, что продемонстрировано в блоке *catch*.

3.2.3. Ключевое слово throws

В некоторых случаях, более целесообразно обрабатывать исключение не в том методе, где оно возникло, а в том, который его вызвал. В таких случаях, в описании метода необходимо объявить (предупредить), что он может вызывать некоторое исключение. Это делается с помощью специального ключевого слова throws.

Пусть метод *getAllScores()* отвечает за чтение результатов игры из файла.

Так как метод *read()* может вызывать исключение *IOException*, нужно или обрабатывать его, или объявить его в описании метода. В примере ниже объявлено, что метод *getAllScores()* может вызывать исключение *IOException*:

```
class MySuperGame{
      void getAllScores() throws IOException{
      file.read();
      }
}
```

3.2.4. Ключевое слово finally

Когда исключение передано, выполнение метода направляется по нелинейному пути. Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе. Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм *finally*.

Ключевое слово *finally* создаёт блок кода, который будет выполнен после завершения блока *try/catch*, но перед кодом, следующим за ним. Блок будет выполнен, независимо от того, передано исключение или нет. Оператор *finally* не обязателен, однако каждый оператор *try* требует наличия либо catch, либо *finally*.

```
try {
      // блок кода, где отслеживаются ошибки
}
catch (тип_исключения_1 exceptionObject) {
      // обрабатываем ошибку
}
catch (тип_исключения_2 exceptionObject) {
      // обрабатываем ошибку
}
finally {
      // код, который нужно выполнить после завершения блока try
}
```

3.2.5. Обработка исключения с помощью конструкции try-catch

Если метод выбросил исключение, он должен предполагать, что исключение будет “поймано” и устранено.

Чтобы увидеть, как ловятся исключения нужно понять концепцию критического блока. **Критический блок** - секция кода, которая может произвести исключение и за которым следует код, обрабатывающий это исключение.

Если вы находитесь внутри метода, и вы выбросили исключение (или другой метод, вызванный вами внутри этого метода, выбросил исключение), такой метод перейдет в процесс бросания. Если вы не хотите быть выброшенными из метода, вы можете установить специальный блок внутри такого метода для поимки исключения. Он называется **блок проверки**, потому что вы “проверяете” ваши различные методы, вызываемые здесь. Блок проверки - это обычный блок, которому предшествует ключевое слово try:

```
try {
  // Код, который может сгенерировать исключение
}
```

При обработке исключений все "тонкие" места программы помещаются в блок проверки и все исключения ловятся в одном месте. Это означает, что код становится намного легче для написания и легче для чтения, поскольку цель кода - не смешиваться с проверкой ошибок.

Выбрасывание исключения должно где-то заканчиваться. Это “место” - **обработчик исключения**, и есть один обработчик для каждого типа исключения, которые вы хотите поймать. Обработчики исключений следуют сразу за блоком проверки и объявляются ключевым словом catch:

```
try {
  // Код, который может сгенерировать исключение
} catch(Type1 id1) {
  // Обработка исключения Type1
} catch(Type2 id2) {
  // Обработка исключения Type2
} catch(Type3 id3) {
  // Обработка исключения Type3
}
// и так далее...
```

Каждое *catch*-предложение (обработчик исключения) как меленький метод, который принимает один и только один аргумент определенного типа. Объекты (id1, id2 и так далее) могут быть использованы внутри обработчика, как аргумент метода. Иногда вы нигде не используете идентификатор, потому что тип исключения дает вам достаточно информации, чтобы разобраться с исключением, но идентификатор все равно должен быть.

Обработчики должны располагаться прямо после блока проверки. Если выброшено исключение, механизм обработки исключений идет охотится за первым обработчиком с таким аргументом, тип которого совпадает с типом исключения. Затем происходит вход в предложение catch, и рассматривается обработка исключения. Поиск обработчика, после остановки на предложении *catch*, заканчивается. Выполняется только совпавшее предложение catch; это не как инструкция switch, в которой вам необходим *break* после каждого *case*, чтобы предотвратить выполнение оставшейся части.

Обратите внимание, что внутри блока проверки несколько вызовов различных методов может генерировать одно и тоже исключение, но вам необходим только один обработчик.

Есть две основные модели в теории обработки исключений. При **прерывании** (которое поддерживает Java и C++), предполагается, что ошибка критична и нет способа вернуться туда, где возникло исключение. Кто бы ни выбросил исключение, он решил, что нет способа спасти ситуацию, и он *не хочет* возвращаться обратно.

Альтернатива называется **возобновлением** - это означает, что обработчик исключения может что-то сделать для исправления ситуации, а затем повторно вызовет придирчивый метод, предполагая, что вторая попытка будет удачной. Если вы хотите возобновления, это означает, что вы все еще надеетесь продолжить выполнение после обработки исключения. В этом случае ваше исключение больше похоже на вызов метода, в котором вы должны произвести настройку ситуации в Java, после чего возможно возобновление. (То есть, не выбрасывать исключение; вызвать метод, который исправит проблему.) Альтернатива этому - поместить блок *try* внутри цикла *while*, который производит повторный вход в блок *try*, пока не будет получен удовлетворительный результат.

Исторически программисты используют операционные системы, которые поддерживают обработку ошибок с возобновлением, в конечном счете, заканчивающуюся использованием прерывающего кода и пропуском возобновления. Так что, хотя возобновление на первый взгляд кажется привлекательнее, оно не так полезно на практике. Вероятно, главная причина - это *соединение* таких результатов: ваш обработчик часто должен знать, где брошено исключение и содержать не характерный специфический код для места выброса. Это делает код трудным для написания и ухода, особенно для больших систем, где исключения могут быть сгенерированы во многих местах.

Можно создать обработчик, ловящий любой тип исключения. Для этого необходимо перехватить исключение базового типа *Exception* (есть другие типы базовых исключений, но *Exception* - это базовый тип, которому принадлежит фактически вся программная активность):

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

Этот код поймает любое исключение, поэтому его нужно помещать в конце списка обработчиков для предотвращения перехвата любого обработчика исключения, который мог управлять течением.

Основные методы класса Exception

String getMessage( )	Получает подробное 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 этого существуют другие методы, наследуемые от базового типа **Throwable Object** (базовый тип для всего). Один из них, который может быть удобен для исключений, это **getClass**(), который возвращает объектное представление класса этого объекта. Вы можете опросить у объекта этого Класса его имя с помощью **getName**( ) или **toString**( ). Вы также можете делать более изощренные вещи с объектом Класса, которые не нужны в обработке ошибок.

Приведем пример использования основных методов **Exception**:

```
public class ExceptionMethods {
  public static void main(String[] args) {
      try {
        throw new Exception("Это мое исключение");
      } catch(Exception e) {
        System.err.println("Поймали 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:

```
Поймали исключение
e.getMessage():  Это мое исключение
e.getLocalizedMessage():  Это мое исключение
e.toString(): java.lang.Exception:
    Это мое исключение
e.printStackTrace():
java.lang.Exception:  Это мое исключение
 at ExceptionMethods.main(ExceptionMethods.java:7)
java.lang.Exception:
    Это мое исключение
 at ExceptionMethods.main(ExceptionMethods.java:7)
```

В этом примере методы обеспечивают больше информации — каждый из них дополняет предыдущий.

Таким образом, имея на вооружении данные методы мы всегда можем точно определить, где и почему произошло исключение и обработать его.

3.2.6 Работа с файлами. Ввод-вывод

Класс File и его методы

Создание хорошей системы ввода/вывода (I/O) является одной из наиболее сложных задач для разработчиков языка.

Доказательством этому служит наличие множества различных подходов. Сложность задачи видится в охвате всех возможностей. Не только различный исходный код и виды ввода/вывода, с которыми вы можете общаться (файлы, консоль, сетевые соединения), но так же вам необходимо общаться с ними большим числом способов (последовательный, в случайном порядке, буферный, бинарный, посимвольный, построчный, пословный и т.п.).

Рассмотрим утилиты, помогающие в обработке директории файлов.

Класс **File** имеет обманчивое имя — вы можете подумать, что он ссылается на файл, но это не так. Он может представлять либо *имя* определенного файла, либо *имя* набора файлов в директории. Если это набор файлов, вы можете опросить набор с помощью метода **list( )**, который вернет массив **String**. Это удобно, потому что число элементов фиксировано, и если вам нужен список другой директории, вы просто создаете другой объект **File**. Фактически, “FilePath” был бы лучшим именем для класса.

Класс **File** может использоваться для создания каталога или дерева каталогов. Также можно узнать свойства файлов (размер, дату последнего изменения, режим чтения/записи), определить к какому типу (файл или каталог) относится объект **File**, удалить файл. У класса очень много методов. Вот некоторые из них:

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

Приведем пример использования некоторых из этих методов:

```
File fl = new File("/example");
System.out.println ("Имя файла:" + fl .getName());
System.out.println ("Путь:" + fl.getPath());
System.out.println ("Полный путь:" + fl.getAbsolutePath());
System.out.println ("Родительский каталог:" + fl.getParent());
System.out.println (fl.exists() ? "существует" : "не существует");
System.out.println (fl.canWrite() ? "можно записывать" : "нельзя записывать");
System.out.println (fl.canRead() ? "можно читать" : "нельзя читать");
System.out.println ("is" + ("Директория? "+(fl.isDirectory() ? "да": "
нет")));
System.out.println (fl.isFile() ? "обычный файл" : "не обычный файл");
System.out.println ("Последняя модификация файла:" + fl.lastModified());
System.out.println ("Размер файла:" + fl.length() + " Bytes");
```

Упражнение 3.2.1

Создайте текстовый файл с именем **«testfile.txt»** на диске и запишите в него информацию о себе: ФИО, дата рождения. Используя методы класса **File,** получите информацию о созданном файле как показано выше.

Вывод информации. Класс PrintWriter

Класс **PrintWriter** очень близок к стандартным возможностям языка C. Методы print() и println() определены для всех стандартных типов (последний добавляет перевод строки в конец вывода), а метод printf() очень похож на соответствующую процедуру языка C. Например, можно написать:

```
PrintWriter writer = new PrintWriter(System.out);
writer.printf("%x", 255);
```

и в консоль будет выведено **FF** (255 в шестнадцатеричном представлении).

Объекты **PrintWriter** обычно работают быстрее стандартного **System.out**. Так как этот класс является оберткой для потоков, с помощью него легко организовать вывод в файл:

```
try {
      File file = new File("d:/testfiles.txt");
      PrintWriter writer = new PrintWriter(new FileWriter(file));
      writer.printf("%x", 255); //Записываем текст в файл
      writer.close(); // Закрываем файл
} catch (IOException e) {
      e.printStackTrace();
}
```

В этом примере создается файл по имени, затем создается поток **FileWriter**, и наконец, он оборачивается в объект **PrintWriter**. Обратите особое внимание на то, что после записи каких либо данных в файл мы должны его закрыть, только после этого действия данные запишутся в файл.

Упражнение 3.2.2

Используя **PrintWriter**, выведите в файл **«testfile.txt»** значение числа 255 в десятичной (%d), восьмеричной (%o) и шестнадцатеричной (%x) системе счисления. Для перехода на новую строку используйте спецификатор %n.

Ввод информации. Класс Scanner

Начиная с версии **1.5**, в языке Java появились преимущества ООП с точки зрения ввода информации. Это было связано с появлением класса Scanner в пакете java.util. Для каждого из базовых типов имеется пара методов: **hasNextT()** говорит, можно ли далее прочесть элемент данного типа **T**, в то время как **nextT()** этот элемент пытается считать. Например, метод **nextInt()** считывает очередной int, а метод **hasNextDouble()** возвращает истину или ложь в зависимости от того, есть ли в потоке 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 обязательно вызывать метод **close()**, который закроет используемый за основной источник поток.

Задание 3.2.1

Реализация посимвольного сравнения двух файлов или страниц в интернете. Выводить требуется все отличающиеся символы в произвольном формате. Если символов очень много нужно вывести только часть и количество различий.

Минипроект

В игре, разработанной в минипроекте темы 3.1 организовать считывание массива ситуаций из файла.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Кравцовой Марии Владимировне.