

Обработка исключений

Обсуждается используемый в Java механизм *обработки исключений*. Исключение в Java — это объект, который описывает исключительное состояние, возникшее в каком-либо участке программного кода. Когда возникает исключительное состояние, создается объект класса `Exception`. Этот объект пересылается в метод, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вручную» для того, чтобы сообщить о некоторых нештатных ситуациях.

Основы

К механизму обработки исключений в Java имеют отношение 5 ключевых слов: — **try**, **catch**, **throw**, **throws** и **finally**. Схема работы этого механизма следующая. Вы пытаетесь (**try**) выполнить блок кода, и если при этом возникает ошибка, система возбуждает (**throw**) исключение, которое в зависимости от его типа вы можете перехватить (**catch**) или передать умалчиваемому (**finally**) обработчику.

Ниже приведена общая форма блока обработки исключений.

```
try {
    // блок кода
}
catch (ТипИсключения1 e) {
    // обработчик исключений типа ТипИсключения1
}
catch (ТипИсключения2 e) {
    // обработчик исключений типа ТипИсключения2
    throw(e) // повторное возбуждение исключения
}
finally {
}
```

Типы исключений

В вершине иерархии исключений стоит класс **Throwable**. Каждый из типов исключений является подклассом класса **Throwable**. Два непосредственных наследника класса **Throwable** делят иерархию подклассов исключений на две различные ветви.

Один из них — класс **Exception** — используется для описания исключительных ситуаций, которые должны перехватываться программным кодом пользователя.

Другая ветвь дерева подклассов **Throwable** — класс **Error**, который предназначен для описания исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

Неперехваченные исключения

Объекты-исключения автоматически создаются исполняющей средой Java в результате возникновения определенных исключительных состояний. Например, очередная наша программа содержит выражение, при вычислении которого возникает деление на нуль.

```
class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
```

Вот вывод, полученный при запуске нашего примера.

C:\> java Exc0

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Обратите внимание на тот факт что типом возбужденного исключения был не **Exception** и не **Throwable**. Это подкласс класса **Exception**, а именно: **ArithmeticException**, поясняющий, какая ошибка возникла при выполнении программы. Вот другая версия того же класса, в которой возникает та же исключительная ситуация, но на этот раз не в программном коде метода **main**.

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    } }
```

Вывод этой программы показывает, как обработчик исключений исполняющей системы Java выводит содержимое всего стека вызовов.

C:\> java Exc1

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

Методы класса Throwable

Блок **catch** может включать разные операторы. Очень часто можно увидеть **catch (Exception e) {**

```
showStatus(e.getMessage());
```

```
}
```

Как видно – здесь подразумевается что **e** – это объект, один из членов которого есть метод **getMessage()**. Этот метод возвращает тип **String**.

Все исключения которые можно перехватить являются подклассами класса **Throwable** (пакет `java.lang`). Класс **Throwable** для них является суперклассом.

Методы класса **Throwable**, определенные с модификатором `private`

String getMessage() – возвращает детализированное сообщение

void printStackTrace() – печать содержимого стека трасировки.

Содержимое стека – перечисление всех методов, прерванных в момент генерации исключения.

String toString() – возвращает описание;

Throwable fillInStackTrace() – заполняет содержимое стека – обычно если надо сгенерировать повторное исключение

try и catch

Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово **try**. Сразу же после **try**-блока помещается блок **catch**, задающий тип исключения которое вы хотите обрабатывать.

```
class Exc2 {
    public static void main(String args[]) {
        try {
            int d = 0;
            int a = 42 / d;
        }
        catch (ArithmeticException e) {
            System.out.println("division by zero");
        }
    }
}
```

Целью большинства хорошо сконструированных **catch**-разделов должна быть обработка возникшей исключительной ситуации и приведение переменных программы в некоторое разумное состояние — такое, чтобы программу можно было продолжить так, будто никакой ошибки и не было (в нашем примере выводится предупреждение – `division by zero`).

Несколько разделов catch

В некоторых случаях один и тот же блок программного кода может возбуждать исключения различных типов. Для того, чтобы обрабатывать

подобные ситуации, Java позволяет использовать любое количество **catch**-разделов для **try**-блока. Наиболее специализированные классы исключений должны идти первыми, поскольку ни один подкласс не будет достигнут, если поставить его после суперкласса. Следующая программа перехватывает два различных типа исключений, причем за этими двумя специализированными обработчиками следует раздел **catch** общего назначения, перехватывающий все подклассы класса **Throwable**.

```
class MultiCatch {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        }
        catch (ArithmeticException e) {
            System.out.println("div by 0: " + e);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("array index oob: " + e);
        }
    }
}
```

Этот пример, запущенный без параметров, вызывает возбуждение исключительной ситуации деления на нуль. Если же мы зададим в командной строке один или несколько параметров, тем самым установив *a* в значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива **ArrayIndexOutOfBoundsException**. Ниже приведены результаты работы этой программы, запущенной и тем и другим способом.

```
C:\> java MultiCatch
a = 0
div by 0: java.lang.ArithmeticException: / by zero
C:\> java MultiCatch 1
a = 1
array index oob:
java.lang.ArrayIndexOutOfBoundsException: 42
```

Вложенные операторы try

Операторы **try** можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возбужденному исключению, стек будет развернут на одну ступень выше, и в поисках подходящего обработчика будут проверены разделы **catch** внешнего оператора **try**. Вот пример, в котором два оператора **try** вложены друг в друга посредством вызова метода.

```
class MultiNest {
    static void procedure() {
        try {
            int c[] = { 1 };
            c[42] = 99;
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("array index oob: " + e);
        }
    }
    public static void main(String args[]) {
        try {
            int a = args.length();
            System.out.println("a = " + a);
            int b = 42 / a;
            procedure();
        }
        catch (ArithmeticException e) {
            System.out.println("div by 0: " + e);
        }
    }
}
```

throw

Оператор **throw** используется для возбуждения исключения «вручную». Для того, чтобы сделать это, нужно иметь объект подкласса класса **Throwable**, который можно либо получить как параметр оператора **catch**, либо создать с помощью оператора **new**. Ниже приведена общая форма оператора **throw**.

throw ОбъектТипаThrowable;

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок **try** проверяется на наличие соответствующего возбужденному исключению обработчика **catch**. Если

такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов **try**, и так до тех пор пока либо не будет найден подходящий раздел **catch**, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор **throw** возбуждает исключительную ситуацию, после чего то же исключение возбуждается повторно — на этот раз уже кодом перехватившего его в первый раз раздела **catch**.

```
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        }
        catch (NullPointerException e) {
            System.out.println("caught inside demoproc");
            throw e;
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        }
        catch (NullPointerException e) {
            System.out.println("recaught: " + e);
        }
    }
}
```

В этом примере обработка исключения проводится в два приема. Метод **main** создает контекст для исключения и вызывает **demoproc**. Метод **demoproc** также устанавливает контекст для обработки исключения, создает новый объект класса **NullPointerException** и с помощью оператора **throw** возбуждает это исключение. Исключение перехватывается в следующей строке внутри метода **demoproc**, причем объект-исключение доступен коду обработчика через параметр **e**. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощью оператора **throw**, в результате чего оно передается обработчику исключений в методе **main**. Ниже приведен результат, полученный при запуске этого примера.

```
C:\> java ThrowDemo
```

```
caught inside demoproc
```

```
recaught: java.lang.NullPointerException: demo
```

throws

Если метод способен возбуждать исключения, которые он сам не обрабатывает, он должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений.

Для задания списка исключений, которые могут возбуждаться методом, используется ключевое слово **throws**. Если метод в явном виде (т.е. с помощью оператора **throw**) возбуждает исключение соответствующего класса, тип класса исключений должен быть указан в операторе **throws** в объявлении этого метода. С учетом этого наш прежний синтаксис определения метода должен быть расширен следующим образом:

тип имя_метода (список аргументов) throws
список_исключений {}

Ниже приведен пример программы, в которой метод `procedure` пытается возбудить исключение, не обеспечивая ни программного кода для его перехвата, ни объявления этого исключения в заголовке метода. Такой программный код не будет оттранслирован.

```
class ThrowsDemo1 {
    static void procedure() {
        System.out.println("inside procedure");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        procedure();
    }
}
```

Для того, чтобы мы смогли оттранслировать этот пример, нам придется сообщить транслятору, что `procedure` может возбуждать исключения типа **IllegalAccessException** и в методе `main` добавить код для обработки этого типа исключений :

```
class ThrowsDemo {
    static void procedure() throws
        IllegalAccessException {
        System.out.println(" inside procedure");
        throw new IllegalAccessException("demo");
    }

    public static void main(String args[]) {
        try {
            procedure();
        }
    }
}
```

```

    }
    catch (IllegalAccessException e) {
        System.out.println("caught " + e);
    }
}
}

```

Ниже приведен результат выполнения этой программы.

C:\> java ThrowsDemo

inside procedure

caught java.lang.IllegalAccessException: demo

Примеры

Читаем строку с клавиатуры

Вот пример класса, который умеет читать строку с клавиатуры и выводить ее на экран:

```

import java.io.*;
class HelloWorld{
    public static void main(String [] args) throws IOException{
        String s;
        BufferedReader in=new BufferedReader(new
                                InputStreamReader(System.in));
        s=in.readLine(); //Читаем с клавиатуры
        System.out.println("s="+s);
    }
}

```

Что мы тут делаем? Сначала мы подключаем java.io.*. Это сделано для того, чтобы не писать длинные имена. Можно было обойтись и без этого - вот так:

```

...
public static void main(String [] args) throws java.io.IOException{
    ...
    java.io.BufferedReader in=new java.io.BufferedReader(
        new java.io.InputStreamReader(System.in));
    ...

```

Но так слишком длинно, поэтому первую строчку и добавили.

Далее мы должны завести переменную для буферизованного ввода. Т. е. для вывода мы специальной переменной не заводили, а для ввода должны завести:

```

...
    BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
    ...

```

Переменная наша называется **in**.

Ну а потом совсем просто - методом `readLine` мы читаем с клавиатуры, и затем выводим на консоль (экран).

Читаем числа с клавиатуры

вот пример класса, который может читать числа с клавиатуры:

```
import java.io.*;
class NumReader{
    public static void main(String [] args) throws IOException{
        String s;
        BufferedReader in=new BufferedReader(new
            InputStreamReader(System.in));
        s=in.readLine(); //Читаем с клавиатуры
        System.out.println("s= "+s);
        double d=Double.valueOf(s).doubleValue(); //Превращение строки в double
        d++;
        System.out.println("d= "+d);
    }
}
```

Для показа, что мы прочитали именно число, мы его сначала увеличиваем на 1, и только после этого выводим на экран.

Если же нам надо превратить строку в целое, то мы можем использовать такую конструкцию:

```
...
    int d=Integer.valueOf(s).intValue();
...
```

Если внимательно присмотреться, то можно увидеть, что мы тут используем странности - Integer вместо **int** и Double вместо **double**. Но именно так и должно быть. Дело в том, что это – классы оболочки. Класс целых чисел и класс вещественных.

finally

Иногда требуется гарантировать, что определенный участок кода будет выполняться независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода используется ключевое слово **finally**. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела **catch**, блок **finally** будет выполнен до того, как управление перейдет к операторам, следующим за разделом **try**. У каждого раздела **try** должен быть по крайней мере или один раздел **catch** или блок **finally**. Блок **finally** очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода. Ниже приведен пример класса с двумя методами, завершение которых происходит по разным причинам, но в обоих перед выходом выполняется код раздела **finally**.

```
class FinallyDemo {
```

```

static void procA() {
    try {
        System.out.println("inside procA");
        throw new RuntimeException("demo");
    }
    finally {
        System.out.println("procA's finally");
    }
}
static void procB() {
    try {
        System.out.println("inside procB");
        return;
    }
    finally {
        System.out.println("procB's finally");
    }
}
public static void main(String args[]) {
    try {
        procA();
    }
    catch (Exception e) {}
    procB();
}
}

```

В этом примере в методе `procA` из-за возбуждения исключения происходит преждевременный выход из блока **try**, но по пути «наружу» выполняется раздел **finally**. Другой метод `procB` завершает работу выполнением стоящего в **try**-блоке оператора **return**, но и при этом перед выходом из метода выполняется программный код блока **finally**. Ниже приведен результат, полученный при выполнении этой программы.

```

C:\> java FinallyDemo
inside procA
procA's finally
inside procB
procB's finally

```

Подклассы Exception

Только подклассы класса **Throwable** могут быть возбуждены или перехвачены. Простые типы — **int**, **char** и т.п., а также классы, не

являющиеся подклассами **Throwable**, например, **String** и **Object**, использоваться в качестве исключений не могут.

Наиболее общий путь для использования исключений — создание своих собственных подклассов класса **Exception**.

Ниже приведена программа, в которой объявлен новый подкласс класса **Exception**.

```
class MyException extends Exception {
    private int detail;
    MyException(int a) {
        detail = a;
    }
    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("called computer + a + ").");
        if (a > 10)
            throw new MyException(a);
        System.out.println("normal exit.");
    }
    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        }
        catch (MyException e) {
            System.out.println("caught" + e);
        }
    }
}
```

Этот пример довольно сложен. В нем сделано объявление подкласса **MyException** класса **Exception**. У этого подкласса есть специальный конструктор, который записывает в переменную объекта целочисленное значение, и совмещенный метод **toString**, выводящий значение, хранящееся в объекте-исключении. Класс **ExceptionDemo** определяет метод **compute**, который возбуждает исключение типа **MyException**.

Простая логика метода **compute** возбуждает исключение в том случае, когда значение параметра метода больше 10. Метод **main** в защищенном блоке вызывает метод **compute** сначала с допустимым значением, а затем — с недопустимым (больше 10), что позволяет продемонстрировать работу при

обоих путях выполнения кода. Ниже приведен результат выполнения программы.

```
C:\> java ExceptionDemo
called compute(1) .
normal exit.
called compute(20) .
caught MyException[20]
```

```
import java.io.*;
import java.util.*;
class testmsg {
    public static void main(String ar[]) throws Exception {

        Msg m=new Msg();
        //Msg.message("Сообщение");
        m.message("Сообщение DOS - 866");
        m.message("Тююс·хэшх win - 1250");
    };
};

class Msg
{
    static String cp = System.getProperty("console.encoding","Cp866");
    public static void message(String msg) throws Exception {
        msg += "\n";
        byte[] b;
        try { b = msg.getBytes(cp); }
        //catch( UnsupportedEncodingException e )
        catch ( Exception e)
        {
            b = msg.getBytes();    // В случае отсутствия нужной кодировки,
        }
        System.out.write(b);
    }
}
```