Лабораторная работа №7

Обработка исключений

Исключение в Java — это объект, который описывает исключительное состояние (ошибку), 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 клю-чевых 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 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
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 ситуации, 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.

```
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.

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

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.

```
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 а в значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива ArrayIndexOutOf Bounds.

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 ОбъектTunaThrowable;

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.

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.

```
inside procedure
caught java.lang.IllegalAccessException: demo
```

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 времен-ного 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 про-исходит 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.

```
inside procA
procA's finally
inside procB
procB's finally
```

Подклассы Exception

Возможно 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);
}
} }
```

В примере 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 MyExcepton. 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.

```
called compute(1).
normal exit.
called compute(20).
caught MyException[20]
```

Легковесные процессы (потоки)

Использование легковесных процессов, или подпроцессов (multithreading, light-weight processes) — концептуальная парадигма, в которой вы разделяете свою программу на два или несколько процессов, которые могут исполняться одновременно.
Если вы можете разделить свою задачу на независимо выполняющиеся подпроцессы и можете автоматически переключаться с одного подпроцесса, который ждет наступления

события, на другой, которому есть чем заняться, за тот же промежуток времени вы выполните больше работы. Вероятность того, что больше чем одному из подпроцессов одновременно надолго потребуется процессор, мала.
Java использует подпроцессы для того, чтобы сделать среду программирования асинхронной. После того, как подпроцесс запущен, его выполнение можно временно приостановить (*suspend*). Если подпроцесс остановлен (*stop*), возобновить его выполнение невозможно.
Приоритеты подпроцессов — это просто целые числа в диапазоне от 1 до 10 и имеет смысл только соотношения приоритетов различных подпроцессов. Приоритеты же используются для того, чтобы решить, когда нужно остановить один подпроцесс и начать выполнение другого. Это называется *переключением контекста.* Правила просты. Подпроцесс может добровольно отдать управление — с помощью явного системного вызова или при блокировании на операциях ввода-вывода, либо он может быть приостановлен принудительно. В первом случае проверяются все остальные подпроцессы, и управление передается тому из них, который готов к выполнению и имеет самый высокий приоритет. Во втором случае, низкоприоритетный подпроцесс, независимо от того, чем он занят, приостанавливается принудительно для того, чтобы начал выполняться подпроцесс с более высоким приоритетом.
Коль скоро вы разделили свою программу на логические части - *подпроцессы*, вам нужно аккуратно определить, как эти части будут общаться друг с другом. Java предоставляет для этого удобное средство — два подпроцесса могут “общаться” друг с другом, используя методы *wait и notify*. Работать с параллельными подпроцессами в Java несложно. Язык предоставляет явный, тонко настраиваемый механизм управления созданием подпроцессов, переключения контекстов, приоритетов, синхронизации и обмена сообщениями между подпроцессами.

Подпроцесс

Класс Thread инкапсулирует все средства, которые могут вам потребоваться при работе с подпроцессами. При запуске Java-программы в ней уже есть один выполняющийся подпроцесс. Вы всегда можете выяснить, какой именно подпроцесс выполняется в данный момент, с помощью вызова статического метода Thread.currentThread. После того, как вы получите дескриптор подпроцесса, вы можете выполнять над этим подпроцессом различные операции даже в том случае, когда параллельные подпроцессы отсутствуют. В очередном нашем примере показано, как можно управлять выполняющимся в данный момент подпроцессом.

```
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
t.setName("My Thread");
System.out. println("current thread: " + t);
try {
for (int n = 5; n > 0; n--) {
System.out.println(" " + n);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("interrupted");
}
} }
```

В этом примере текущий подпроцесс хранится в локальной переменной t. Затем мы используем эту переменную для вызова метода setName, который изменяет внутреннее имя подпроцесса на “My Thread”, с тем, чтобы вывод программы был удобочитаемым. На

следующем шаге мы входим в цикл, в котором ведется обратный отсчет от 5, причем на каждой итерации с помощью вызова метода Thread.sleep() делается пауза длительностью в 1 секунду. Аргументом для этого метода является значение временного интервала в миллисекундах. Обратите внимание — цикл заключен в try/catch блок. Дело в том, что метод Thread.sleep() может возбуждать исключение InterruptedException. Это исключение возбуждается в том случае, если какому-либо другому подпроцессу понадобится прервать данный подпроцесс. В данном примере мы в такой ситуации просто выводим сообщение о перехвате исключения. Ниже приведен вывод этой программы:

```
current thread: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что в текстовом представлении объекта Thread содержится заданное нами имя легковесного процесса — My Thread. Число 5 — это приоритет подпроцесса, оно соответствует приоритету по умолчанию, “main” — имя группы подпроцессов, к которой принадлежит данный подпроцесс.

Runnable

Не очень интересно работать только с одним подпроцессом, а как можно создать еще один? Для этого нам понадобится другой экземпляр класса Thread. При создании нового объекта Thread ему нужно указать, какой программный код он должен выполнять. Вы можете запустить подпроцесс с помощью любого объекта, реализующего интерфейс Runnable. Для того, чтобы реализовать этот интерфейс, класс должен предоставить определение метода run. Ниже приведен пример, в котором создается новый подпроцесс.

```
class ThreadDemo implements Runnable {
ThreadDemo() {
Thread ct = Thread.currentThread();
System.out.println("currentThread: " + ct);
Thread t = new Thread(this, "Demo Thread");
System.out.println("Thread created: " + t);
t.start();
try {
Thread.sleep(3000);
}
catch (InterruptedException e) {
System.out.println("interrupted");
}
System.out.println("exiting main thread");
}
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("" + i);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("child interrupted");
}
System.out.println("exiting child thread");
}
```

```
public static void main(String args[]) {
new ThreadDemo();
} }
```

Обратите внимание на то, что цикл внутри метода run выглядит точно так же, как и в предыдущем примере, только на этот раз он выполняется в другом подпроцессе. Подпроцесс main с помощью оператора new Thread(this, "Demo Thread") создает новый объект класса Thread, причем первый параметр конструктора — this — указывает, что нам хочется вызвать метод run текущего объекта. Затем мы вызываем метод start, который запускает подпроцесс, выполняющий метод run. После этого основной подпроцесс (main) переводится в состояние ожидания на три секунды, затем выводит сообщение и завершает работу. Второй подпроцесс — “Demo Thread” — при этом по-прежнему выполняет итерации в цикле метода run до тех пор пока значение счетчика цикла не уменьшится до нуля. Ниже показано, как выглядит результат работы этой программы этой программы после того, как она отработает 5 секунд.

```
Thread created: Thread[Demo Thread,5,main]
5
4
3
exiting main thread
2
1
exiting child thread
```

Приоритеты подпроцессов

Если вы хотите добиться от Java предсказуемого независимого от платформы поведения, вам следует проектировать свои подпроцессы таким образом, чтобы они по своей воле освобождали процессор. Ниже приведен пример с двумя подпроцессами с различными приоритетами, которые не ведут себя одинаково на различных платформах. Приоритет одного из подпроцессов с помощью вызова setPriority устанавливается на два уровня выше Thread. NORM_PRIORITY, то есть, умалчиваемого приоритета. У другого подпроцесса приоритет, наоборот, на два уровня ниже. Оба этих подпроцесса запускаются и работают в течение 10 секунд. Каждый из них выполняет цикл, в котором увеличивается значение переменной-счетчика. Через десять секунд после их запуска основной подпроцесс останавливает их работу, присваивая условию завершения цикла while значение true и выводит значения счетчиков, показывающих, сколько итераций цикла успел выполнить каждый из подпроцессов.

```
class Clicker implements Runnable {
int click = 0;
private Thread t;
private boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
} }
public void stop() {
running = false; }
public void start() {
```

```
t.start();
} }
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
}
catch (Exception e) {
}
lo.stop();
hi.stop();
System.out.println(lo.click + " vs. " + hi.click);
} }
```

По значениям, полученным после окончания работы программы можно заключить, что подпроцессу с низким приоритетом достается меньше времени процессора.

Синхронизация

Когда двум или более подпроцессам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), нужно позаботиться о том, чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из подпроцессов. Java для такой синхронизации предоставляет уникальную, встроенную в язык программирования поддержку. В других системах с параллельными подпроцессами существует понятие *монитора.* Монитор — это объект, используемый как защелка. Только один из подпроцессов может в данный момент времени владеть монитором. Когда под-процесс получает эту защелку, говорят, что он *вошел* в монитор. Все остальные подпроцессы, пытающиеся войти в тот же монитор, будут заморожены до тех пор пока подпроцесс-владелец не выйдет из монитора.

У каждого Java-объекта есть связанный с ним неявный монитор, а для того, чтобы войти в него, надо вызвать метод этого объекта, отмеченный ключевым словом **synchronized**. Для того, чтобы выйти из монитора и тем самым передать управление объектом другому подпроцессу, владелец монитора должен всего лишь вернуться из синхронизованного метода.

Взаимодействие подпроцессов

В Java имеется элегантный механизм общения между подпроцессами, основанный на методах **wait, notify** и **notifyAll**. Эти методы реализованы, как final-методы класса Object, так что они имеются в любом Java-классе. Все эти методы должны вызываться только из синхронизованных методов. Правила использования этих методов очень просты:

• wait — приводит к тому, что текущий подпроцесс отдает управление и переходит в режим ожидания — до тех пор пока другой под-процесс не вызовет метод notify с тем же объектом.

• notify — выводит из состояния ожидания первый из подпроцессов, вызвавших wait с данным объектом.

• notifyAll — выводит из состояния ожидания все подпроцессы, вызвавшие wait с данным объектом.

Сводка функций программного интерфейса легковесных процессов

Ниже приведена сводка всех методов класса Thread, обсуждавшихся в этой главе.

Методы класса

Методы класса — это статические методы, которые можно вызывать непосредственно с именем класса Thread.

currentThread

Статический метод currentThread возвращает объект Thread, выполняющийся в данный момент.

yield

Вызов метода yield приводит к тому, что исполняющая система переключает контекст с текущего на следующий доступный подпроцесс. Это один из способов гарантировать, что низкоприоритетные подпроцессы когда-нибудь получат управление.

sleep(int n)

При вызове метода sleep исполняющая система блокирует текущий подпроцесс на n миллисекунд. После того, как этот интервал времени закончится, подпроцесс снова будет способен выполняться. В большинстве исполняющих систем Java системные часы не позволяют точно выдерживать паузы короче, чем 10 миллисекунд.

Методы объекта

start

Метод start говорит исполняющей системе Java, что необходимо создать системный контекст подпроцесса и запустить этот подпроцесс. После вызова этого метода в новом контексте будет вызван метод run вновь созданного подпроцесса. Вам нужно помнить о том, что метод start с данным объектом можно вызвать только один раз.

run

Метод run — это тело выполняющегося подпроцесса. Это — единственный метод интерфейса Runnable. Он вызывается из метода start после того, как исполняющая среда выполнит необходимые операции по инициализации нового подпроцесса. Если происходит возврат из метода run, текущий подпроцесс останавливается.

stop

Вызов метода stop приводит к немедленной остановке подпроцесса. Это — способ мгновенно прекратить выполнение текущего подпроцесса, особенно если метод выполняется в текущем подпроцессе. В таком случае строка, следующая за вызовом метода stop, никогда не выполняется, поскольку контекст подпроцесса “умирает” до того, как метод stop возвратит управление. Более аккуратный способ остановить выполнение подпроцесса — установить значение какой-либо переменной-флага, предусмотрев в методе run код, который, проверив состояние флага, завершил бы выполнение подпроцесса.

suspend

Метод suspend отличается от метода stop тем, что метод приостанавливает выполнение подпроцесса, не разрушая при этом его системный контекст. Если выполнение подпроцесса приостановлено вызовом suspend, вы можете снова активизировать этот подпроцесс, вызвав метод resume.

resume

Метод resume используется для активизации подпроцесса, приостановленного вызовом suspend. При этом не гарантируется, что после вызова resume подпроцесс немедленно начнет выполняться, поскольку в этот момент может выполняться другой более высокоприоритетный процесс. Вызов resume лишь делает подпроцесс способным выполняться, а то, когда ему будет передано управление, решит планировщик.

setPriority(int p)

Метод setPriority устанавливает приоритет подпроцесса, задаваемый целым значением передаваемого методу параметра. В классе Thread есть несколько предопределенных приоритетов-констант: MIN_PRIORITY, NORM_PRIORITY и MAX_PRIORITY, соответствующих соответственно значениям 1, 5 и 10. Большинство пользовательских приложений должно выполняться на уровне NORM_PRIORITY плюс-минус 1. Приоритет фоновых заданий, например, сетевого ввода-вывода или перерисовки экрана, следует

устанавливать в MIN_PRIORITY. Запуск подпроцессов на уровне MAX_PRIORITY требует осторожности. Если в подпроцессах с таким уровнем приоритета отсутствуют вызовы sleep или yield, может оказаться, что вся исполняющая система Java перестанет реагировать на внешние раздражители.

getPriority

Этот метод возвращает текущий приоритет подпроцесса — целое значение в диапазоне от 1 до 10.

setName(String name)

Метод setName присваивает подпроцессу указанное в параметре имя. Это помогает при отладке программ с параллельными подпроцессами. Присвоенное с помощью setName имя будет появляться во всех трассировках стека, которые выводятся при получении интерпретатором неперехваченного исключения.

getName

Метод getName возвращает строку с именем подпроцесса, установленным с помощью вызова setName.

Задание 1.

Создать класс, а в нем метод с параметром – целым числом. Этот метод возводит число-параметр в квадрат, если оно положительное и вызывает исключение, если параметр меньше или равен нулю. Указанное исключение необходимо создать самостоятельно и вызывать его в методе с помощью *throw* и *throws*. Исключение должно обрабатываться в методе `main` и выдавать сообщение *Это мое исключение!*. При обработке исключения в блоке *finally* выводить сообщение *Исключения отработаны!*

Задание 2.

Создать два потока, каждый из которых выводит на экран целые числа от 0 до 9 с задержкой в секунду. Затем для первого потока увеличить время задержки на 250 миллисекунд, а для другого – уменьшить его на такую же величину. Проанализировать результат.