1. Обработка исключений

Исключение — это аварийное состояние, которое возникает в кодовой последовательности во время выполнения. Другими словами, исключение — это ошибка времени выполнения. В машинных языках, не поддерживающих обработку исключений, ошибки должны быть проверены и обработаны вручную — обычно с помощью кодов ошибки, и т.д. Такой подход довольно сложен и трудоемок. Обработка исключений в Java избегает этих проблем и переносит управление обработкой ошибок времени выполнения в объектно-ориентированное русло.

1.1. Основные принципы обработки исключений

Исключение в языке Java — это объект, который описывает исключительную (т.е. ошибочную) ситуацию, произошедшую в некоторой части кода. Когда исключительная ситуация возникает, создается объект, представляющий это исключение, и *"вбрасывается"* в метод, вызвавший ошибку. В свою очередь, метод может выбрать, обрабатывать ли исключение самому или передать его куда-то еще. В любом случае, в некоторой точке исключение *"захватывается"* и обрабатывается. Исключения могут генерироваться исполнительной системой Java, или ваш код может сгенерировать их "вручную". Выбрасываемые исключения касаются фундаментальных ошибок, которые нарушают ограничения среды выполнения или правила языка Java. Исключения, сгенерированные вручную, обычно используются, чтобы сообщить вызывающей программе о некоторой аварийной ситуации.

Обработка исключений в Java управляется с помощью пяти ключевых слов — try, catch, throw, throws и finally. Опишем кратко, как они работают.

Программные операторы, которые нужно контролировать относительно исключений, заключаются в блок try. Если в блоке try происходит исключение, говорят, что оно *выброшено* (thrown) этим блоком. Код может *перехватить* (catch) это исключение (используя оператор catch) и обработать его некоторым способом. Исключения, генерируемые исполнительной (run-time) системой Java, выбрасываются автоматически. Для "ручного" выброса исключения используется ключевое слово throw. Любое исключение, которое выброшено из метода, следует определять с помощью ключевого слова throws,

размещаемого в заголовочном предложении определения метода. Любой код, который обязательно должен быть выполнен перед возвратом из try-блока, размещается в finally-блоке, указанном в конце блочной конструкции try{... }-catch{. .. }-finally{ . ..}.

Общая форма блока обработки исключений:

```
try{
// блок кода для контроля над ошибками
}

catch (ExceptionType1 exOb)   {
// обработчик исключений для ExceptionType1
}

catch (ExceptionType2 exOb)   {
// обработчик исключений для ExceptionType2
}
//...
 [finally{
// блок кода для обработки перед возвратом из try блока
}]
```

Здесь ExceptionType — тип исключения, которое возникло; exOb — объект этого исключения, finally-блок — не обязателен. Далее описывается, как следует применять эту структуру.

1.2. Типы исключений

Все типы исключений являются подклассами встроенного класса Throwable. Таким образом, Throwable представляет собой вершину иерархии классов исключений. Непосредственно ниже Throwable находятся два подкласса, которые разделяют исключения на две различные ветви. Одна ветвь возглавляется классом Exception. Этот класс используется для исключительных состояний, которые должны перехватывать программы пользователя. Это также класс, в подклассах которого вы будете создавать ваши собственные заказные типы исключений. У Exception имеется важный подкласс, называемый RuntimeException. Исключения этого типа для ваших программ определены автоматически и включают такие события, как деление на нуль, недопустимая индексация массива и т. п.

Другую ветвь возглавляет класс Error, определяющий исключения, перехват которых вашей программой при нормальных обстоятельствах не ожидается. Исключения типа Error применяются исполнительной системой Java для указания ошибок, имеющих отношение непосредственно к среде времени выполнения. Пример подобной ошибки — переполнение стека. Эта глава не содержит описание исключений типа Error из-за того, что они обычно создаются в ответ на

катастрофические отказы, которые, как правило, не могут обрабатываться вашей программой.

Неотловленные исключения

Прежде чем узнать, как обрабатывать исключения в своей программе, полезно посмотреть, что произойдет, если их не обрабатывать. Следующая маленькая программа включает выражение, которое преднамеренно вызывает ошибку деления на нуль.

Программа 46. Встроенная обработка исключений

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

Когда исполнительная система Java обнаруживает попытку деления на нуль, она создает новый объект исключения и затем *выбрасывает* его. Это заставляет выполнение Exc0 остановиться, потому что, как только исключение окажется выброшенным, оно должно быть *захвачено* обработчиком исключений, и причем — немедленно. В представленном примере мы не снабдили набор имеющихся обработчиков исключений своим собственным вариантом, так что исключение захватывается обработчиком, заданным исполнительной системой Java по умолчанию. Любое исключение, которое не захвачено программой, будет в конечном счете выполнено обработчиком по умолчанию. Этот обработчик отображает (на экран) строку, описывающую исключение, печатает трассу стека от точки, в которой произошло исключение, и завершает программу.

Вот какой вывод генерирует предложенный пример, когда он выполняется стандартным Java-интерпретатором из JDK:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Exc0.main(Exc0.java:4)
```

В трассу стека (`at Exc0.main(Exc0.java:4)`) включены следующие элементы: имя класса (Exc0), имя метода (main), имя файла (Exe0.java) и номер строки (4). Кроме того, из первой строки сообщения видно, что тип выброшенного исключения является подклассом Exception с именем ArithmeticException, которое более определенно описывает тип произошедшей ошибки. Как будет обсуждено позже, Java поставляет несколько встроенных типов исключений, которые соответствуют различным видам ошибок, генерируемых во время выполнения.

Трасса стека всегда показывает последовательность вызовов методов, которые привели к ошибке. Далее представлена другая версия предыдущей программы, которая представляет ту же самую ошибку, но в отдельном от main() методе. Класс Exc1 добавлен в программу 46:

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

Результирующая трасса стека (полученная от обработчика исключений по умолчанию) показывает, как отображается полный стек вызовов:

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

Видно, что основанием стека является строка 7 метода main, которая является обращением к методу subroutine(), генерирующему исключение в строке 4. Стек вызовов весьма полезен для отладки, потому что он довольно точно отражает последовательность шагов, которые привели к ошибке.

1.3. Использование операторов try и catch

Хотя умалчиваемый обработчик исключений исполнительной системы Java полезен для отладки, программисту обычно хочется обрабатывать исключение самостоятельно. Это обеспечивает два преимущества: во-первых, позволяет фиксировать ошибку и, во-вторых, предохраняет программу от автоматического завершения. Большинство пользователей было бы смущено (мягко говоря), если бы ваша программа прекращала выполнение и печатала трассу стека всякий раз, когда произошла ошибка. К счастью, эту ситуацию весьма просто предотвратить.

Для того чтобы отслеживать и обрабатывать ошибку времени выполнения, просто включите код, который нужно контролировать, внутрь блока try. Сразу после блока try укажите catch-блок, определяющий тип исключения, которое нужно перехватить, и его обработчик. Чтобы проиллюстрировать, как легко это можно сделать, приведем программу, включающую блок try и блок catch, который обрабатывает исключение типа ArithmeticException, генерируемое ошибкой "деление на нуль":

Программа 47. Программирование обработки исключений

```
class Exc2 {
   public static void main(String arcs[]) {
      int d, a;
      try {              // Контролировать блок кода
         d = 0;
         a = 42 / d;
         System.out.println ("Это не будет напечатано.");
      }
      catch (ArithmeticException e) {       // Перехватить ошибку
                                            // деления на нуль
         System.out.println("Деление на нуль.");
      }
      System.out.println("После оператора catch.");
   }
}
```

Эта программа генерирует следующий вывод:

```
Деление на нуль.
После оператора catch.
```

Обратите внимание, что обращение к println() внутри блока try никогда не выполняется. Как только исключение выброшено, управление программой передается из блока try в блок catch. Размещенный по-другому блок catch не вызывается, так что управление (выполнением) никогда не возвращается из блока catch блоку try. Таким образом, строка "Это не будет напечатано." никогда не выведется на экран. Сразу после выполнения оператора catch программное управление продолжается со строки, следующей за полным механизмом try/catch.

Таким образом, try и его catch-оператор формируют небольшой программный модуль (точнее — пару связанных блоков). Область видимости catch-утверждения ограничена ближайшим предшествующим утверждением try. Оператор catch не может захватывать исключение, выброшенное другим try-оператором (кроме случая вложенных try-операторов, кратко описанных далее). Операторы, которые контролируются утверждением try, должны быть окружены фигурными скобками (т. е. они должны быть внутри блока). Нельзя использовать try с одиночным утверждением (без скобок).

Целью хорошо сконструированного catch-предложения должно быть разрешение исключительной ситуации с последующим продолжением выполнения программы, как будто ошибка никогда не возникала. Например, в следующей программе каждая итерация цикла for получает два случайных целых числа. Они делятся друг на друга, и их частное используется для деления значения 12 345. Конечный

результат помещается в переменную a. Если любая операция деления приводит к ошибке деления на нуль, она перехватывается, значение сбрасывается в нуль, и программа продолжается.

Программа 48. Продолжение вычислений после обработки исключения

```
// Обработать исключение и продолжить.
import java.util.Random;
class HandleError {
   public static void main(String args[]) {
      int a = 0, b = 0, c = 0;
      Random r = new Random(); // Объект для генерации случайных чисел
      for(int i = 0; i < 32000; i++) {
         try {
            b = r.nextInt();
            c = r.nextInt();
            a = 12345 / (b / c);
         }
         catch (ArithmeticException e) {
            System.out.println("Деление на нуль.");
            a = 0;                // Сбросить в нуль и продолжить
         }
         System.out.println("a:  " + a);
      }
   }
}
```

1.4. Отображение описания исключения

Класс Throwable переопределяет метод toString() (определенный в классе object) так, чтобы он возвращал строку, содержащую описание исключения. Вы можете отображать это описание методом println(), просто передавая ему исключение как аргумент. Например, блок catch в предыдущей программе может быть переписан так:

```
catch (ArithmeticException e) {
   System.out.println("Исключение:  " + e);
   a = 0;                // Сбросить в нуль и продолжить
}
```

Если этой версией catch заменить имеющуюся в программе и выполнить программу стандартным Java-интерпретатором из JDK, каждая ошибка деления на нуль отобразит следующее сообщение:

```
Исключение: Java.lang.ArithmeticException: / by zero
```

Хотя в данном контексте это не имеет особого значения, в других обстоятельствах способность отображать описание исключения может оказаться достаточно ценной — особенно, когда вы экспериментируете с исключениями или для отладки.

1.5. Русификация консольных приложений

Для кодировки символов в Java используется Unicode, при этом для один символ кодируется используется двумя или более байтами памяти. Строки символов Java состоят, соответсвенно, из многобайтовых символов. При выводе символов в потоки, например, при использовании println() происходит преобразование символов в байты и эти байты выводятся в поток. Эти преобразования выполняются в соответствии с установками по умолчанию, поэтому русские буквы из программы не появляются в консольном окне. В программах, работающих под Windows, используется кодовая страница Cp1251, а в выходном консольном окне – кодовая страница Cp866. Поэтому для правильного вывода русских букв нужно выполнить их преобразование. Вариант такого преобразования представлен в следующей программе.

Программа 49. Перекодировка строк

В состав программы входит класс WinToDos, имеющий один статический метод MakeDos(), который преобразует строку, полученную в качестве аргумента, к кодовой странице Cp866 и возвращает ее.

```
// Файл WinDos.java
import java.io.*;
class WinToDos{
   static String MakeDos(String s)          // Преобразование строки в Cp866
   {
      try{
         byte b[] = s.getBytes("Cp866");    // Преобразование строки
                                            // в массив байтов
         return new String(b,"Cp1251");     // Преобразование массива байтов
                                            // в строку
      }
      catch(UnsupportedEncodingException e){
         return s;
      }
   }
}

public class WinDos {
   static public void main(String args[]) throws IOException{
      System.out.println("АБВГДЕЁЖЗабвгдеёжз " + 321);
      System.out.println(WinToDos.MakeDos("АБВГДЕЁЖЗабвгдеёжз ") + 123);
   }
}
```

При запуске из среды Eclipse программа выводит:

```
АБВГДЕЁЖЗабвгдеёжз 321
ЂЃ‚ѓ„…р†‡ ЎЎЈ¤Ґс¦§ 123
```

Мы видим, что руский текст, заданный непосредственно в программном коде правильно выводится в выходной окно, создаваемое средой Eclipse.

Результат запуска программы из командной строки показан на рис. 9.1.

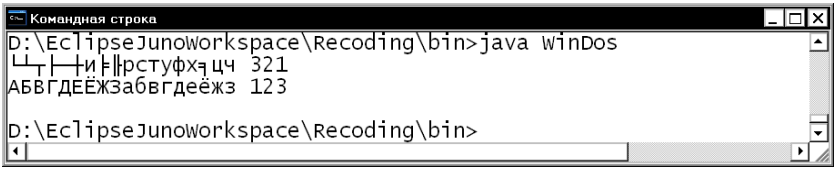

Рис. 9.1. Вывод программы при запуске из командной строки

Здесь русский текст, заданный в программе выводится неправильно, но после его преобразования методом MakeDos() получается правильный результат.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	Ђ 0402	Ѓ 0403	‚ 201A	ѓ 0453	„ 201E	… 2026	† 2020	‡ 2021	€ 20AC	‰ 2030	Љ 0409	‹ 2039	Њ 040A	Ќ 040C	Ћ 040B	Џ 040F
90	ђ 0452	‘ 2018	’ 2019	“ 201C	” 201D	• 2022	– 2013	— 2014		™ 2122	љ 0459	› 203A	њ 045A	ќ 045C	ћ 045B	џ 045F
A0	NBSP 00A0	Ў 040E	ў 045E	Ј 0408	¤ 00A4	Ґ 0490	¦ 00A6	§ 00A7	Ё 0401	© 00A9	Є 0404	« 00AB	¬ 00AC	- 00AD	® 00AE	Ї 0407
B0	° 00B0	± 00B1	І 0406	і 0456	ґ 0491	µ 00B5	¶ 00B6	· 00B7	ё 0451	№ 2116	є 0454	» 00BB	ј 0458	Ѕ 0405	ѕ 0455	ї 0457
C0	А 0410	Б 0411	В 0412	Г 0413	Д 0414	Е 0415	Ж 0416	З 0417	И 0418	Й 0419	К 041A	Л 041B	М 041C	Н 041D	О 041E	П 041F
D0	Р 0420	С 0421	Т 0422	У 0423	Ф 0424	Х 0425	Ц 0426	Ч 0427	Ш 0428	Щ 0429	Ъ 042A	Ы 042B	Ь 042C	Э 042D	Ю 042E	Я 042F
E0	а 0430	б 0431	в 0432	г 0433	д 0434	е 0435	ж 0436	з 0437	и 0438	й 0439	к 043A	л 043B	м 043C	н 043D	о 043E	п 043F
F0	р 0440	с 0441	т 0442	у 0443	ф 0444	х 0445	ц 0446	ч 0447	ш 0448	щ 0449	ъ 044A	ы 044B	ь 044C	э 044D	ю 044E	я 044F

Рис . 9.2. Кодовая страница 1251

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	А 0410	Б 0411	В 0412	Г 0413	Д 0414	Е 0415	Ж 0416	З 0417	И 0418	Й 0419	К 041A	Л 041B	М 041C	Н 041D	О 041E	П 041F
90	Р 0420	С 0421	Т 0422	У 0423	Ф 0424	Х 0425	Ц 0426	Ч 0427	Ш 0428	Щ 0429	Ъ 042A	Ы 042B	Ь 042C	Э 042D	Ю 042E	Я 042F
A0	а 0430	б 0431	в 0432	г 0433	д 0434	е 0435	ж 0436	з 0437	и 0438	й 0439	к 043A	л 043B	м 043C	н 043D	о 043E	п 043F
B0	▒ 2591	▓ 2592	▓ 2593	┃ 2502	┤ 2524	╡ 2561	╢ 2562	╖ 2556	╕ 2555	╣ 2563	║ 2551	╗ 2557	╝ 255D	╜ 255C	╛ 255B	┐ 2510
C0	└ 2514	┴ 2534	┬ 252C	├ 251C	─ 2500	┼ 253C	╞ 255E	╟ 255F	╚ 255A	╔ 2554	╩ 2569	╦ 2566	╠ 2560	═ 2550	╬ 256C	╧ 2567
D0	╨ 2568	╤ 2564	╥ 2565	╙ 2559	╘ 2558	╒ 2552	╓ 2553	╫ 256B	╪ 256A	┘ 2518	┌ 250C	▒ 2588	▄ 2584	▌ 258C	▐ 2590	▀ 2580
E0	р 0440	с 0441	т 0442	у 0443	ф 0444	х 0445	ц 0446	ч 0447	ш 0448	щ 0449	ъ 044A	ы 044B	ь 044C	э 044D	ю 044E	я 044F
F0	Ё 0401	ё 0451	Є 0404	є 0454	Ї 0407	ї 0457	Ў 040E	ў 045E	° 00B0	∙ 2219	· 00B7	√ 221A	№ 2116	¤ 00A4	■ 25A0	NBSP 00A0

Рис. 9.3. Кодовая страница 866

Русские буквы 'А', 'Б', 'В', … имеют в кодовой странице Cp1251 коды 0xC0, 0xC1, 0xC2,… При выводе в консольное окно появляются символы '└', '┴', '├',… с этими кодами из кодовой страницы Cp866.

Русские буквы 'А', 'Б', 'В', … в кодовой таблице Cp866 имеют коды 0x80, 0x81, 0x82,… и в окне вывода в среде Eclipse появляются символы 'Ъ', 'Ѓ', '‚',… , имеющие указанные коды в кодовой таблице Cp1251.

1.6. Множественные операторы catch

В некоторых случаях на одном участке кода может возникнуть более одного исключения. Чтобы обрабатывать этот тип ситуации, необходимо определить несколько операторов catch, каждый — для захвата своего типа исключения. Когда выбрасывается исключение,

каждый catch-оператор просматривается по порядку и первый, чей тип соответствует типу возникшего исключения, выполняется. После того как этот catch-оператор выполнится, другие — обходятся, и выполнение продолжается после блока try/catch. Следующий пример отлавливает исключение двух различных типов:

Программа 50. Несколько обработчиков исключений

В программе анализируется количество аргументов, заданных в командной строке при запуске программы. Если аргументы не заданы возникнет ошибка деления на нуль, иначе будет обращение к несуществующему элементу массива.

```
// Файл MultiCatch.java
import java.io.*;
class WinToDos{
   static String MakeDos(String s)
   {
      try{
         byte b[] = s.getBytes("Cp866");    // Преобразование строки в
                                            // массив байтов
         return new String(b, "Cp1251");    // Преобразование массива байтов
                                            // в строку
      }
      catch(UnsupportedEncodingException e){
         return s;
      }
   }
}

// Демонстрация множественных catch-операторов.
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 };      // Массив из одного элемента
         c[42] = 99;           // Недопустимый индекс
      }
      catch(ArithmeticException e) {
         System.out.println(WinToDos.MakeDos("Деление на нуль:  ") + e);
      }
      catch(ArrayIndexOutOfBoundsException e) {
         System.out.println(WinToDos.MakeDos(
                        "Индекс элемента массива c:  ") + e);
      }
      System.out.println(WinToDos.MakeDos("После блока try/catch."));
   }
}
```

Эта программа выбросит исключение "деление на нуль", если она будет запускаться без параметров командной строки, так как

переменная a будет равна нулю. Этой ситуации не возникнет, если вы укажете аргумент в командной строке, устанавливающий в a что-то большее, чем нуль. Но это вызовет исключение ArrayIndexOutOf-BoundsException, так как целочисленный массив c имеет длину 1, тогда как программа пытается назначить некоторое значение его сорок второму элементу c[42].

На рис.9.4 показан результат запуска програмы без аргументов командной строки. Возникла ошибка деления на нуль. На рис. 9.5. показана ошибка выхода за границы массива.

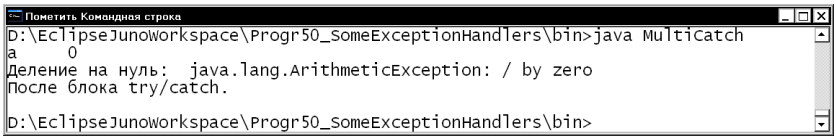

Рис. 9.4. Результаты запуска при незаданных аргументах командной строки

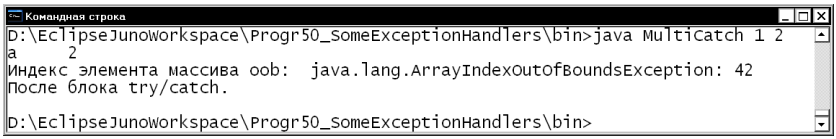

Рис. 9.5. Запуск программы с аргументами командной строки

Когда вы используете множественные catch-операторы, важно помнить, что в последовательности catch-предложений подклассы исключений должны следовать перед любым из их суперклассов. Это происходит потому, что предложение catch, которое использует суперкласс, будет перехватывать исключения как своего типа, так и любого из своих подклассов. Таким образом, подкласс никогда не был бы достигнут, если бы он следовал после своего суперкласса. Кроме того, в Java недостижимый код — ошибка. Например, рассмотрим следующую программу:

```
/*   Эта программа содержит ошибку.
Подкласс должен следовать раньше своего суперкласса в серии catch-
операторов. Если это не так, то в результате будет создаваться недостижимый
код и соответствующий тип ошибки времени выполнения.  */
class SuperSubCatch {
   public static void main(String args[j) {
      try {
         int a = 0;
         int b = 42 / a;
      }
      Catch(Exception e) {
         System.out.printin("Генерация исключения catch.");
      }
```

```
/* Этот catch никогда не будет достигнут из-за того, что ArithmeticException
является подклассом Exception. */
      catch (ArithmeticException e) {       // ОШИБКА. Оператор недостижим
         System.out.printin("Недостижимый оператор.");
      }
   }
}
```

Если вы попытаетесь откомпилировать данную программу, то примете сообщение об ошибке, заявляющее, что второй catch-оператор недостижим.

Так как ArithmeticException — подкласс Exception, первый catch-оператор обработает все ошибки, основанные на Exception, включая и ArithmeticException. Это означает, что второй catch-оператор никогда не будет выполняться. Чтобы устранить проблему, измените порядок операторов catch.

Программа 51. Пакет русификации

Создадим пакет, содержащий класс для преобразования кодировки руских букв. Для этого создадим проект Russification, при этом будет создана папка с таким именем, и в составе этого проекта создадим пакет RusWinToDos (рис.9.6)

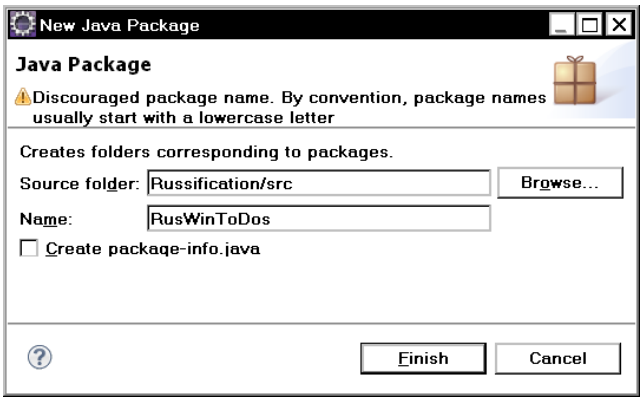

Рис. 9.6. Создание пакета для преобразования русских букв

В состав пакета включим класс WinDos (рис.9.7)

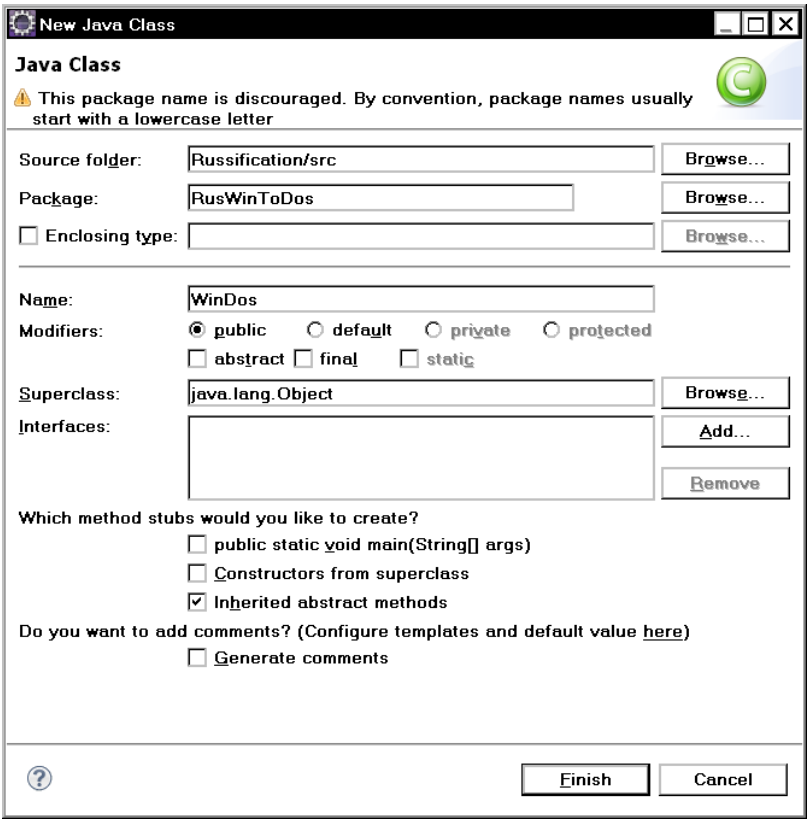

Рис. 9.7. Добавление в пакет класса

Далее приведен текст класса:

```
// Файл WinDos.java
package RusWinToDos;

import java.io.UnsupportedEncodingException;

   public class WinDos {
      static String MakeDos(String s)
      {
         try{
            byte b[] = s.getBytes("Cp866");    // Преобразование строки
                                               // в массив байтов
            return new String(b, "Cp1251");    // Преобразование массива
                                               // байтов в строку
         }
         catch(UnsupportedEncodingException e){
            return s;
         }
```

```
        }
}
```

Пример использования созданного пакета RusWinToDos приводится в следующей программе.

Вложенные операторы try

Операторы try могут быть вложенными. То есть один try-оператор может находиться внутри блока другого оператора try. При входе в блок try контекст соответствующего исключения помещается в стек. Если внутренний оператор try не имеет catch-обработчика для специфического исключения, стек раскручивается, и просматривается следующий catch-обработчик try-оператора (для поиска соответствия с типом исключения). Процесс продолжается до тех пор, пока не будет достигнут подходящий catch-оператор, или пока все вложенные операторы try не будут исчерпаны. Если согласующегося оператора catch нет, то исключение обработает исполнительная система Java. Пример, который использует вложенные операторы try:

Программа 52. Вложенные try

Чтобы можно было использовать пакет, созданный в другом проекте, нужно это указать в свойствах проекта, выполнив команду Project, Properties, Java Build Path и в список Required projects on the build path добавить проект, содержащий нужный пакет (рис.)

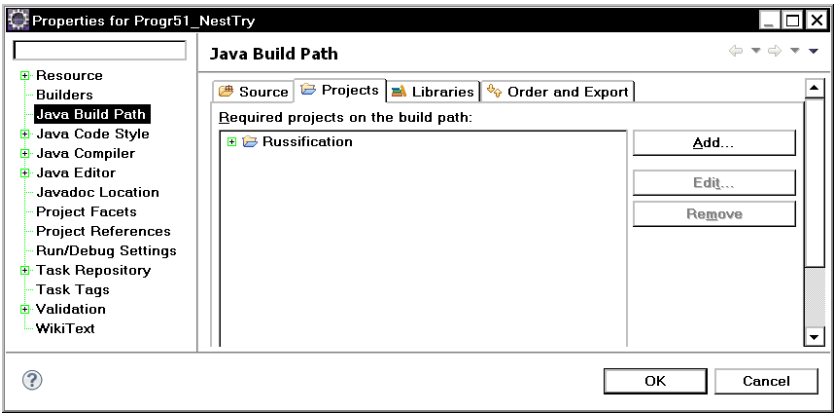

Рис. 9.8. Добавление проектов, от которых зависит данный проект

```
// Файл NestTry.java
import RusWinToDos.*;
// Демонстрация множественных catch-операторов.
public class NestTry {
```

```
   public static void main(String[] args){
      try {
         int a = args.length;
/* Если нет аргументов командной строки,
следующий оператор будет генерировать
исключение деления на нуль. */
         int b = 42 / a;
         System.out.println("a = " + a);
         try {                  // Вложенный try-блок
/* Если используется один аргумент командной строки,
то следующий код будет генерировать
исключение деления на нуль. */
            if(a == 1)
               a = a / (a - a);       // Деление на нуль
/* Если используется два аргумента командной строки,
то генерируется исключение выхода за границу массива.  */
            if (a == 2) {
               int c[] = { 1 };
               c[42] = 99;        // Генерировать исключение
            }                     // выхода за границу массива
         }                        // Вложенный try
         catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(WinDos.MakeDos(
                               "Индекс выходит за границу массива: ") + e);
         }
      }  // Внешний try
      catch(ArithmeticException e) {
         System.out.println(WinDos.MakeDos("Деление на нуль: ") + e);
      }
   }
}
```

Пример демонстрирует вложение одного try-блока в другой. При выполнении программы без аргументов командной строки исключение деления на нуль генерируется внешним блоком try. Выполнение программы с одним аргументом командной строки генерирует исключение деления на нуль внутри вложенного блока try. Так как внутренний блок не захватывает это исключение, он пересылает его внешнему блоку try, где оно и обрабатывается. Если вы запустите программу с двумя аргументами командной строки, то во внутреннем блоке try генерируется исключение нарушения границы массива.

Запусти данную программу из командной строки два раза без аргументов в командной строке и с аргументами. Следующие протоколы выполнения иллюстрируют каждый из этих случаев:

```
C:\>java NestTry
Деление на нуль: Java.lang.ArithmeticException:  / by zero

C:\>java NestTry One
a = 1
Деление на нуль: Java.lang.ArithmeticException:  / by zero
```

```
C:\>java NestTry One Two
a = 2
Индекс выходит за границу массива:
Java.lang.ArraylndexOutOfBoundsException: 42
```

Вложение инструкций try может происходить менее очевидными способами, когда вложенный try-блок организован в отдельном методе и выполняется через вызов этого метода во внешнем блоке. Ниже показана предыдущая программа, переписанная так, чтобы вложенный блок try был перемещен внутрь метода nesttry ():

Программа 53. Вложенные try в методе

```
/* try-операторы можно неявно вкладывать
через вызовы методов.  */
import java.io.*;
class WinToDos{
   static String MakeDos(String s)
   {
      try{
         byte b[] = s.getBytes("Cp866");    // Преобразование строки в
                                            // массив байтов
         return new String(b, "Cp1251");    // Преобразование массива байтов
                                            // в строку
      }
      catch(UnsupportedEncodingException e){
         return s;
      }
   }
}

class MethNestTry {
   static void nesttry(int a) {
      try {              // Вложенный try-блок
/* Если используется один аргумент командной строки, то следующий код будет
генерировать исключение деления на нуль. */
         if(a == 1)
            a = a / (a - a);   // деление на нуль
/* Если используется два аргумента командной строки,
то генерируется исключение выхода за границу массива.  */
         if(a == 2) {
            int c[] = { 1 };
            c[42] = 99;        // Генерировать исключение
         }                     // выхода за границу массива
      }
      catch(ArrayIndexOutOfBoundsException e) {
         System.out.println(WinToDos.MakeDos(
                           "Индекс выходит за границу массива:  ") + e);
      }
   }
   public static void main(String args[]) {
      try {
         int a = args.length;
```

```
/* Если нет аргументов командной строки,
следующий оператор будет генерировать
исключение деления на нуль. */
          int b = 42 / a;
          System.out.println ("a = " + a);
          nesttry(a);
       }
       catch(ArithmeticException e) {
          System.out.println(WinToDos.MakeDos("Деление на нуль:  ") + e);
       }
    }
}
```

Вывод этой программы идентичен выводу предыдущего примера.

1.7. Оператор throw

До сих пор вы захватывали только исключения, которые выбрасывались исполнительной системой Java. Однако программа может сама явно выбрасывать исключения, используя оператор throw. Общая форма оператора throw такова:

```
throw ThrowableInstance;
```

Здесь ThrowableInstance должен быть объектом типа Throwable или подкласса Throwable. Простые типы, такие как int или char, а также не-Throwable-классы (типа string и object) не могут использоваться как исключения. Имеется два способа получения Throwable-объекта: использование параметра в предложении catch или создание объекта с помощью операции new.

После оператора throw поток выполнения немедленно останавливается, и любые последующие операторы не выполняются. Затем просматривается ближайший включающий блок try с целью поиска оператора catch, который соответствует типу исключения. Если соответствие отыскивается, то управление передается этому оператору. Если нет, то просматривается следующее включение оператора try и т. д. Если соответствующий catch не найден, то программу останавливает обработчик исключений, заданный по умолчанию, и затем выводится трасса стека.

Пример программы, которая создает и выбрасывает исключение (обработчик, который захватывает исключение, перебрасывает его во внешний обработчик):

Программа 54. Демонстрация оператора throw

```
// Файл TrowDemo.java
// Демонстрирует throw.
class ThrowDemo {
   static void demoproc() {
```

```
      try {
         throw new NullPointerException("demo");
      }
      catch(NullPointerException e) {
         System.out.println("Захват внутри demoproc.");
         throw e;              // Повторный выброс исключения
      }
   }
   public static void main(String args[]) {
      try {
         demoproc();
      }
      catch(NullPointerException e1) {
         System.out.println("Повторный захват:   " + e1);
      }
   }
}
```

Эта программа получает две возможности иметь дело с одной и той же ошибкой. Сначала main() устанавливает контекст исключения и затем вызывает demoproc(). Потом метод demoproc() устанавливает другой контекст — для обработки особых ситуаций и немедленно выбрасывает новый экземпляр исключения NullPointerException, который захватывается на следующей строке. Далее это исключение выбрасывается повторно. Итоговый вывод этой программы:

```
Захват внутри   demoproc.
Повторный захват: Java.lang.NullPointerException: demo
```

Программа также иллюстрирует, как можно создавать один из стандартных объектов исключений. Уделите побольше внимания следующей строке:

```
throw new NullPointerException("demo");
```

Здесь new используется для создания экземпляра класса NullPointerException.

Все встроенные исключения времени выполнения имеют два конструктора — один без параметра, а другой — со строчным параметром. Когда используется вторая форма, аргумент определяет строку, описывающую исключение. Данная строка отображается на экран, когда объект указывается в качестве аргумента методами print() или printin(). Ее можно также получить с помощью вызова метода getMessage(), который определен в классе Throwable.

1.8. Методы с ключевым словом throws

Если метод способен к порождению исключения, которое он не обрабатывает, он должен определить свое поведение так, чтобы вызывающие методы могли сами предохранять себя от данного

исключения. Это обеспечивается включением предложения throws в заголовок объявления метода. Предложение throws перечисляет типы исключений, которые метод может выбрасывать. Это необходимо для всех исключений, кроме исключений типа Error, RuntimeException или любых их подклассов. Все другие исключения, которые метод может выбрасывать, должны быть объявлены в предложении throws. Если данное условие не соблюдено, то произойдет ошибка времени компиляции.

Общая форма объявления метода, которое включает предложение throws:

```
type method-name (parameter-list) throw exception-list {
      // тело метода
}
```

Здесь exception-list — список разделенных запятыми исключений, которые метод может выбрасывать.

Ниже показан пример неправильной программы, пытающейся выбросить исключение, которое она не перехватывает. Поскольку программа не определяет предложение throws, чтобы объявить этот факт, программа не будет компилироваться.

```
// Эта программа содержит ошибку и не будет компилироваться.
class ThrowsDemo {
   static void throwOne() {
      System.out.println("Внутри throwOne.");
      throw new IllegalAccessException("demo");
   }
   public static void main(String args[]) {
      throwOne();
   }
}
```

Чтобы сделать этот пример компилируемым, требуется внести два изменения. Во-первых, нужно объявить, что throwone() выбрасывает исключение IllegalAccessException. Во-вторых, main() должен определить оператор try/catch, который захватывает исключение. Исправленный пример выглядит так:

Программа 55. Указание исключений в методе (throws)

```
// Файл ThrowsDemo.java
// Теперь эта программа корректна.
class ThrowsDemo {
   static void throwOne() throws IllegalAccessException {
      System.out.println("Внутри throwOne.");
      throw new IllegalAccessException("demo");
   }
   public static void main(String args[]) {
      try {
```

```
            throwOne();
        }
        catch (IllegalAccessException e) {
            System.out.println("Выброс " + e);
        }
    }
}
```

Вывод, сгенерированный выполнением этой программы:

```
Внутри throwOne.
Выброс Java.lang.IllegalAccessException: demo
```

1.9. Блок finally

Когда исключение выбрасывается, выполнение метода имеет довольно неровный, нелинейный путь, который изменяет нормальное прохождение потока через метод. В зависимости от того, как кодирован метод, исключение может вызвать даже преждевременный выход из него. В некоторых случаях это могло бы стать проблемой. Например, если метод открывает файл для ввода и закрывает его для вывода, то вы вряд ли захотите, чтобы код, закрывающий файл был обойден механизмом обработки исключений. Для реализации этой возможности и предназначено ключевое слово finally.

finally определяет блок кода, выполняющийся после того, как блок try/catch завершился и перед кодом, следующим за блоком try/catch. Блок finally будет выполняться независимо от того, было ли выброшено исключение или нет. Если исключение было выброшено, конструкции блока finally будут обрабатываться, даже если нет catch-оператора, соответствующего исключению. Предложение finally выполняется каждый раз, когда метод собирается вернуться к вызывающей программе изнутри блока try/catch (через непойманное исключение или явный оператор return), причем выполнение происходит непосредственно перед возвратом из метода. Это может быть полезно для закрытия дескрипторов файла и освобождения любых других ресурсов, которые могли быть распределены в начале метода, с намерением избавиться от них перед возвратом. Предложение finally необязательно. Однако каждый оператор try требует по крайней мере одного предложения catch или finally.

Пример программы, которая демонстрирует три метода, выполняющие выход различными способами (причем ни один не обходится без выполнения своих finally-предложений):

Программа 56. Использование finally

```
// Файл FinallyDemo.java
// Демонстрирует finally.
```

```
class FinallyDemo {

   static void procA() {
      try {
         System.out.println("Внутри procA");
         throw new RuntimeException("demo");   // Выход из метода через
                                               // исключение.
      }
      finally {
         System.out.println("finally для procA ");
      }
   }
// Возврат изнутри try-блока.
   static void procB () {
      try {
         System.out.println("Внутри procB");
         return;
      }
      finally {
         System.out.println("finally для procB ");
      }
   }
// Нормальное выполнение try-блока.
   static void procC() {
      try {
         System.out.println("Внутри procC");
      }
      finally {
         System.out.println("finally procC");
      }
   }
   public static void main(String args[]) {
      try {
         procA();
      }
      catch (Exception e) {
         System.out.println("Исключение выброшено");
      }
      procB();
      procC();
   }
}
```

В этом примере, procA() преждевременно выходит из блока try, выбрасывая исключение. Перед самым выходом выполняется предложение finally. Оператор try метода procB() выходит с помощью оператора return. Здесь finally запускается перед возвратом из procB(). В методе procC() оператор try выполняется нормально, без ошибки. Однако блок finally все же реализуется.

Замечание

Если блок finally связывается с блоком try, то при возврате из блока try блок finally будет выполняться всегда.

Вывод, сгенерированный предшествующей программой:

```
Внутри procA
finally для procA
Исключение выброшено
Внутри procB
finally для procB
Внутри procC
finally для procC
```

1.10. Встроенные исключения Java

Внутри стандартного пакета java.lang определено несколько классов исключений. Некоторые из них использовались в предыдущих примерах. Наиболее общие из этих исключений — это подклассы стандартного типа RuntimeException. Так как java.lang неявно импортирован во все Java-программы, большинство исключений, производных от RuntimeException, доступны автоматически. Более того, их не нужно включать в throws-список любого метода. В языке Java они называются *неконтролируемыми исключениями,* потому что компилятор не проверяет, выбрасывает или обрабатывает метод эти исключения. Неконтролируемые исключения, определенные в java.lang (как подклассы Exception), перечислены в табл. 10. Табл. 11 содержит описание исключений, определенных в java.lang, которые должны быть включены в throws-список метода, если данный метод может генерировать одно из указанных исключений и не обрабатывает его сам. Они называются *контролируемыми исключениями* и являются подклассами класса RuntimeException. Кроме того, в Java определены несколько других типов исключений, которые относятся к различным библиотекам его классов.

Таблица 10 Подклассы неконтролируемых исключений

Исключение	Значение
ArithmeticException	Арифметическая ошибка типа деления на нуль
ArrayIndexOutOfBoundsException	Индекс массива находится вне границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора, типа ожидания на разблокированном потоке
IllegalStateException	Среда или приложение находятся в

	некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
SecurityException	Попытка нарушить защиту
StringIndexOutOfBoundsException	Попытка индексировать вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

Таблица 11. Контролируемые исключения, определенные в java. lang

Исключение	Значение
ClassNotFoundException	Класс не найден
CloneNotSupportedException	Попытка клонировать объект, который не реализует интерфейс Cloneable
IllegalAccessException	Доступ к классу отклонен
UnsupportedOperationException	Встретилась неподдерживаемая операция
InstantiationException	Попытка создавать объект абстрактного класса или интерфейса
InterruptedException	Один поток был прерван другим потоком
NoSuchFieldException	Требуемое поле не существует
NoSuchMethodException	Требуемый метод не существует

1.11. Создание собственных подклассов исключений

Хотя встроенные исключения языка Java обрабатывают наиболее общие ошибки, вам, вероятно, захочется создать свои собственные типы исключений с целью обработки ситуаций, специфических для ваших приложений. Для этого просто определите подкласс Exception (который, конечно, является подклассом Throwable). На самом деле ваши

подклассы не должны ничего реализовывать — само существование их в системе типов позволяет использовать их как исключения.

Класс Exception не определяет никаких собственных методов, а наследует эти методы от класса Throwable. Таким образом, всем исключениям, даже тем, что вы создаете сами, доступны методы Throwable. Список их представлен в табл. 12. Кроме того, вы можете переопределить один или несколько этих методов в создаваемых вами классах исключений.

Таблица 12 Методы, определенные в Throwable

Метод	Описание
Throwable fillInStackTrace()	Возвращает Throwable-объект, который содержит полную трассу стека. Этот объект может быть выброшен повторно
String getLocalizedMessage()	Возвращает локализованное описание исключения
String getMessage()	Возвращает описание исключения
void printStackTrace()	Отображает трассу стека
void printStackTrace (Printstream stream)	Посылает трассу стека указанному потоку
void printStackTrace (PrintWriter stream)	Посылает проекцию прямой стека указанному потоку
string toString()	Возвращает string-объект, содержащий описание исключения. Этот метод вызывается из println() при выводе Throwable-объекта

Следующий пример объявляет новый подкласс Exception и затем использует его, чтобы сигнализировать об аварийной ситуации в методе. Он переопределяет метод toString(), позволяя отобразить описание исключения с помощью println().

Программа 57. Собственный класс исключений

```
// Файл MyExceptionDemo.java
// Эта программа создает заказной тип исключения.
class MyException extends Exception {
   private int detail;
   MyException(int a) {
      detail = a;
   }
   public String toString() {
      return "MyException[" + detail + "]";
   }
}
```

```
class MyExceptionDemo {
   static void compute(int a) throws MyException {
      System.out.println("Вызван compute(" + a + ")");
      if (a > 10)
         throw new MyException(a);
      System.out.println("Нормальный выход");
   }
   public static void main(String args[]) {
      try {
         compute(1);
         compute(20);
      }
      catch (MyException e) {
         System.out.println("Выброшено " + e);
      }
   }
}
```

Этот пример определяет подкласс Exception с именем MyException. Этот подкласс имеет только конструктор и перегруженный метод tostring(), который отображает значение исключения. Класс ExceptionDemo определяет метод, названный compute(), который выбрасывает объект MyException. Исключение выбрасывается, когда целый параметр метода compute() больше 10. Метод main() устанавливает обработчик исключений для MyException, а затем вызывает compute() с допустимым значением параметра (меньшим 10) и недопустимым, чтобы показать оба варианта работы программы. Вот результат:

```
Вызван compute(1)
Нормальный выход
Вызван compute(20)
Выброшено MyException[20]
```

1.12. Использование исключений

Обработка исключений обеспечивает мощный механизм управления комплексными программами, обладающими множеством динамических характеристик времени выполнения. Важно представлять механизм try-throw-catch, как достаточно ясный способ обработки ошибок и необычных граничных условий в логике программы. Как и большинство программистов, вы, вероятно, привыкли возвращать код ошибки, когда метод терпит неудачу. Если вы программируете на языке Java, то нужно отказаться от этой привычки. Когда произойдет отказ метода, пусть он сам выбросит исключение. Это более ясный способ обработки режимов отказа.