



# Содержание

<b>Предисловие</b>	<b>19</b>
Структура книги	20
Условные обозначения, принятые в книге	20
Об авторах	21
Благодарности	22
Об изображении на обложке книги	22
Ждем ваших отзывов!	23
<b>Часть I. Язык</b>	<b>25</b>
<b>Глава 1. Соглашения об именовании</b>	<b>27</b>
Акронимы	27
Имена аннотаций	27
Имена классов	28
Имена констант	28
Имена перечислений	28
Имена параметров обобщенного типа	29
Имена переменных экземпляра и статических переменных	29
Имена интерфейсов	30
Имена методов	30
Имена пакетов	30
Имена модулей	31
Имена параметров и локальных переменных	31
<b>Глава 2. Лексические элементы</b>	<b>33</b>
Символы в коде ASCII и Юникоде	33
Отображаемые символы в коде ASCII	34

Неотображаемые символы в коде ASCII	35
Компактные строки	35
Комментарии	36
Ключевые слова	37
Идентификаторы	38
Разделители	39
Операции	39
Литералы	41
Логические литералы	41
Символьные литералы	41
Целочисленные литералы	42
Числовые литералы с плавающей точкой	43
Строковые литералы	44
Нулевые литералы	45
Управляющие последовательности символов	45
Знаки денежных единиц в Юникоде	46
<b>Глава 3. Основные типы данных</b>	<b>49</b>
Примитивные типы	49
Литералы примитивных типов	50
Сущности с плавающей точкой	53
Операции с особыми сущностями	54
Числовое продвижение примитивных типов	55
Унарное числовое продвижение	55
Бинарное числовое продвижение	56
Особые случаи применения условных операций	57
Классы-оболочки	57
Автоупаковка и распаковка	58
Автоупаковка	58
Распаковка	59
<b>Глава 4. Ссылочные типы</b>	<b>61</b>
Сравнение ссылочных и примитивных типов	62
Стандартные значения	62
Переменные экземпляра и локальные переменные	63
Массивы	64
Преобразование ссылочных типов	65
Расширяющие преобразования	65
Сужающие преобразования	66

Взаимное преобразование примитивных и ссылочных типов	66
Передача ссылочных типов методам	67
Сравнение ссылочных типов	68
Применение операций равенства	68
Применение метода <code>equals()</code>	69
Сравнение строк	69
Сравнение перечислений	71
Копирование ссылочных типов	71
Копирование ссылки на объект	71
Клонирование объектов	71
Распределение памяти и сборка “мусора”	73
<b>Глава 5. Объектно-ориентированное программирование</b>	<b>75</b>
Классы и объекты	75
Синтаксис класса	76
Получение экземпляра класса (или создание объекта)	76
Элементы данных и методы	77
Доступ к свойствам и методам в объектах	77
Перегрузка	78
Переопределение	78
Конструкторы	79
Суперклассы и подклассы	80
Ключевое слово <code>this</code>	82
Списки с переменным числом аргументов	82
Абстрактные классы и методы	84
Абстрактные классы	84
Абстрактные методы	85
Статические элементы данных, методы, константы и инициализаторы	85
Статические элементы данных	85
Статические методы	86
Статические константы	86
Статические инициализаторы	86
Интерфейсы	87
Перечисления	88
Типы аннотаций	89
Встроенные аннотации	89
Аннотации, определяемые разработчиком	90
Функциональные интерфейсы	92

<b>Глава 6. Операторы и блоки</b>	<b>93</b>
Операторы выражений	93
Пустой оператор	94
Блоки	94
Условные операторы	94
Условный оператор <code>if</code>	95
Условный оператор <code>if else</code>	95
Условный оператор <code>if else if</code>	95
Оператор <code>switch</code>	96
Итеративные операторы	97
Оператор цикла <code>for</code>	97
Оператор расширенного цикла <code>for</code>	98
Оператор цикла <code>while</code>	98
Оператор цикла <code>do-while</code>	99
Передача управления	99
Оператор <code>break</code>	99
Оператор <code>continue</code>	100
Оператор <code>return</code>	101
Оператор <code>synchronized</code>	101
Оператор <code>assert</code>	101
Операторы обработки исключений	102
<b>Глава 7. Обработка исключений</b>	<b>103</b>
Иерархия исключений	103
Проверяемые и непроверяемые исключения и ошибки	104
Проверяемые исключения	104
Непроверяемые исключения	105
Ошибки	105
Типичные проверяемые и непроверяемые исключения и ошибки	106
Типичные проверяемые исключения	106
Типичные непроверяемые исключения	107
Типичные ошибки	108
Ключевые слова для обработки исключений	109
Ключевое слово <code>throw</code>	109
Ключевые слова <code>try/catch/finally</code>	110
Оператор <code>try-catch</code>	110

Оператор <code>try-finally</code>	112
Оператор <code>try-catch-finally</code>	113
Оператор <code>try</code> с ресурсами	113
Конструкция для перехвата нескольких исключений	114
Процесс обработки исключений	115
Определение собственного класса исключений	115
Вывод сведений об исключениях	116
Метод <code>getMessage()</code>	116
Метод <code>toString()</code>	117
Метод <code>printStackTrace()</code>	117
<b>Глава 8. Модификаторы Java</b>	<b>119</b>
Модификаторы доступа	120
Остальные модификаторы	121
Кодирование модификаторов	122
<b>Часть II. Платформа</b>	<b>123</b>
<b>Глава 9. Платформа Java, стандартный выпуск</b>	<b>125</b>
Стандартные библиотеки Java SE API	125
Библиотеки языка и утилит	126
Базовые библиотеки	128
Библиотеки интеграции	130
Различные библиотеки пользовательского интерфейса	131
Библиотеки пользовательского интерфейса: JavaFX	132
Библиотеки RMI и CORBA	135
Библиотеки обеспечения безопасности	136
Библиотеки XML	138
<b>Глава 10. Основы разработки</b>	<b>141</b>
Исполняющая среда Java	141
Комплект разработки прикладных программ на Java	141
Структура программы на Java	143
Утилиты командной строки	145
Компилятор Java	145
Интерпретатор Java	147
Упаковщик программ на Java	149
Запуск архивных JAR-файлов на выполнение	150
Параметр <code>classpath</code>	152

<b>Глава 11. Управление памятью</b>	<b>153</b>
Сборщики “мусора”	153
Последовательный сборщик “мусора”	154
Параллельный сборщик мусора	154
Параллельный уплотняющий сборщик “мусора”	155
Сборщик “мусора” с одновременной маркировкой и очисткой	155
Сборщик “мусора” в первую очередь	155
Средства управления памятью	156
Параметры командной строки	157
Изменение размера “кучи” в виртуальной машине JVM	161
Метапространство	162
Взаимодействие со сборкой “мусора”	162
Сборка “мусора” явным образом	162
Ликвидация объектов	163
<b>Глава 12. Основы организации ввода-вывода</b>	<b>165</b>
Стандартные потоки ввода-вывода <code>in</code> , <code>out</code> и <code>err</code>	166
Иерархия основных классов ввода-вывода	166
Чтение и запись файлов	167
Чтение символьных данных из файла	168
Чтение двоичных данных из файла	168
Запись символьных данных в файл	169
Запись двоичных данных в файл	169
Чтение и запись данных в сокетах	170
Чтение символьных данных из сокета	170
Чтение двоичных данных из сокета	171
Запись символьных данных в сокет	171
Запись двоичных данных в сокет	171
Сериализация	172
Сериализация	172
Десериализация	173
Упаковка и распаковка архивных файлов	173
Обращение с архивными ZIP-файлами	173
Обращение с архивными GZIP-файлами	174
<b>Глава 13. Новый механизм ввода-вывода NIO 2.0</b>	<b>175</b>
Интерфейс <code>Path</code>	175
Класс <code>Files</code>	176
Дополнительные возможности	178

<b>Глава 14. Параллелизм</b>	<b>181</b>
Создание потоков исполнения	181
Расширение класса <b>Thread</b>	181
Реализация интерфейса <b>Runnable</b>	182
Состояния потока исполнения	183
Приоритеты потоков исполнения	183
Типичные методы для работы с потоками исполнения	184
Синхронизация	186
Служебные средства параллелизма	187
Исполнители	187
Многопоточные коллекции	189
Синхронизаторы	189
Служебные средства согласования по времени	190
<b>Глава 15. Каркас коллекций Java</b>	<b>191</b>
Интерфейс <b>Collection</b>	191
Реализации	192
Методы из интерфейсов коллекций	193
Алгоритмы из класса <b>Collections</b>	194
Эффективность алгоритмов коллекций	195
Функциональный интерфейс <b>Comparator</b>	196
Удобные фабричные методы	199
<b>Глава 16 Каркас обобщений</b>	<b>201</b>
Обобщенные классы и интерфейсы	201
Конструкторы с обобщениями	203
Принцип подстановки	203
Параметры типа, метасимволы подстановки и ограничения	204
Принцип “извлечь и внести”	205
Обобщенная специализация	206
Обобщенные методы в базовых типах классов	207
<b>Глава 17. Прикладной интерфейс Java Scripting API</b>	<b>209</b>
Языки сценариев	209
Реализация интерпретаторов сценарных языков	209
Встраивание сценариев в программы на Java	210



Вызов методов сценарных языков	211
Доступ к ресурсам Java и управление ими из сценариев	211
Установка сценарных языков и интерпретаторов	212
Установка сценарного языка	212
Установка интерпретатора сценарного языка	213
Проверка правильности установки интерпретатора	214
<b>Глава 18. Прикладной интерфейс API даты и времени</b>	<b>217</b>
Совместимость с устаревшим кодом	218
Региональные календари	219
Календарь по стандарту ISO	219
Основные классы даты и времени	220
Машинный интерфейс	222
Продолжительности и периоды времени	223
Соответствие типов JDBC и XSD	224
Форматирование	224
<b>Глава 19. Лямбда-выражения</b>	<b>227</b>
Основы лямбда-выражений	227
Синтаксис и примеры лямбда-выражений	228
Ссылки на методы и конструкторы	230
Функциональные интерфейсы специального назначения	231
Функциональные интерфейсы общего назначения	231
Ресурсы по лямбда-выражениям	233
Учебные пособия	233
Общедоступные ресурсы	233
<b>Глава 20. Утилита JShell</b>	<b>235</b>
Как приступить к работе с JShell	235
Фрагменты кода	236
Модификаторы	236
Операторы управляющей логики	237
Объявления пакетов	237
Применение JShell	238
Простейшие выражения	238
Зависимости	239
Операторы и блоки кода	239
Объявления классов и методов	241
Просмотр, удаление и видоизменение фрагментов кода	243

Сохранение, загрузка и восстановление состояния	245
Функциональные средства JShell	246
Временные переменные	246
Автозавершение по табуляции	247
Опережающие ссылки	248
Проверяемые исключения	248
Иерархия и области видимости	250
Сводка команд JShell	251
<b>Глава 21. Модульная система на платформе Java</b>	<b>253</b>
Проект Jigsaw	253
Модули в Java	254
Автоматические модули	256
Безымянные модули	256
Доступность на уровне модулей	257
Компиляция модулей	257
Модульный комплект JDK	259
Утилита <code>jdeps</code>	261
Выявление зависимостей в классах	261
Выявление зависимостей от недокументированных классов в JDK	262
Определение модулей	263
Экспорт пакетов	264
Объявление зависимостей	265
Транзитивные зависимости	265
Объявление поставщиков услуг	266
Определение прикладного интерфейса API услуги	267
Реализация прикладного интерфейса API услуги	267
Применение поставщиков услуг	268
Утилита <code>jlink</code>	269
<b>Часть III. Приложения</b>	<b>271</b>
<b>Приложение А. Текущие прикладные интерфейсы API</b>	<b>273</b>
<b>Приложение Б. Сторонние инструментальные средства</b>	<b>277</b>
Средства разработки, конфигурирования и тестирования	277
Библиотеки	281
Интегрированные среды разработки	285
Платформы веб-приложений	286
Языки сценариев, совместимые с JSR 223	288

<b>Приложение В. Основы UML</b>	<b>291</b>
Диаграммы классов	291
Наименование	292
Атрибуты	292
Операции	293
Видимость	293
Диаграммы объектов	294
Графические символы для представления диаграмм	294
Обычные, абстрактные классы и интерфейсы	294
Примечания	295
Пакеты	295
Соединения	296
Индикаторы множественности	296
Имена ролей	297
Отношения между классами	297
Ассоциация	298
Прямая ассоциация	298
Композиция	298
Агрегация	298
Временная ассоциация	299
Обобщение	299
Реализация	299
Диаграммы последовательности	299
Участник (1)	300
Найденное сообщение (2)	300
Синхронное сообщение (3)	300
Ответный вызов (4)	300
Асинхронное сообщение (5)	301
Сообщение, адресованное самому себе (6)	301
Линия жизни (7)	301
Полоса активизации (8)	301
<b>Предметный указатель</b>	<b>303</b>

# Операторы и блоки

Оператор — это отдельная команда, осуществляющая конкретное действие, когда она выполняется интерпретатором Java:

```
GigSim simulator = new GigSim("Let's play guitar!");
```

Операторы в Java могут включать в себя выражения, блоки кода, передачу управления, обработку исключений, переменные, метки, утверждения, а также пустые, условные, итеративные и синхронизированные операторы. В операторах Java употребляются следующие зарезервированные слова: `if`, `else`, `switch`, `case`, `while`, `do`, `for`, `break`, `continue`, `return`, `synchronized`, `throw`, `try`, `catch`, `finally` и `assert`.

## Операторы выражений

Оператор выражения — это такой оператор, который изменяет состояние программы. Любое выражение в Java оканчивается точкой с запятой. К числу операторов выражений относятся: присваивания, префиксные и постфиксные приращения (инкременты), префиксные и постфиксные вычитания (декременты), создание объектов и вызовы методов. Ниже приведены некоторые примеры операторов выражений.

```
isWithinOperatingHours = true;  
++fret; patron++; --glassOfWater; pick--;
```

```
Guitarist guitarist = new Guitarist();  
guitarist.placeCapo(guitar, capo, fret);
```

## Пустой оператор

Пустой оператор не выполняет никакого действия и обозначается точкой с запятой (;) или фигурными скобками {} как пустой блок.

## Блоки

Группа операторов называется просто *блоком* или же *блоком операторов*. Блок операторов заключается в фигурные скобки. Переменные и классы, объявленные в блоке, называются локальными переменными и локальными классами соответственно. Область видимости локальных переменных и локальных классов определяется тем блоком, в котором они объявлены.

Операторы в блоках интерпретируются в той последовательности, в которой они написаны, или в порядке управления выполнением программы. Ниже приведен пример блока операторов.

```
static {  
    GigSimProperties.setFirstFestivalActive(true);  
    System.out.println("First festival has begun");  
    gigsimLogger.info("Simulator started 1st festival");  
}
```

## Условные операторы

Условные операторы `if`, `if else` и `if else if` относятся к категории операторов выбора дальнейшего выполнения программы. Они служат для выполнения других операторов по заданным условиям. Выражение для любого из этих операторов

должно иметь тип `Boolean` или `boolean`. Если это тип `Boolean`, то выполняется распаковка, т.е. автоматическое преобразование типа `Boolean` в тип `boolean`.

## Условный оператор `if`

Условный оператор `if` состоит из выражения и единственного оператора или целого блока операторов, которые выполняются, если в результате вычисления условного выражения получается логическое значение `true`.

```
Guitar guitar = new Guitar();
guitar.addProblemItem("Треснувший гриф");
if (guitar.isBroken()) {
    Luthier luthier = new Luthier();
    luthier.repairGuitar(guitar);
}
```

## Условный оператор `if else`

Если условный оператор `if` дополняется оператором `else`, то первый блок операторов в конструкции `if else` выполняется в том случае, если в результате вычисления условного выражения получается логическое значение `true`. В противном случае выполняется второй блок операторов в конструкции `if else`.

```
CoffeeShop coffeeshop = new CoffeeShop();
if (coffeeshop.getPatronCount() > 5) {
    System.out.println("Play the event.");
} else {
    System.out.println("Go home without pay.");
}
```

## Условный оператор `if else if`

Условный оператор `if else if` обычно применяется в том случае, когда требуется сделать выбор среди нескольких блоков кода. Если же заданные критерии выбора не позволяют

выполнить ни один из этих блоков, то выполняется код в последнем блоке оператора `else`.

```
ArrayList<Song> playList = new ArrayList<>();
Song song1 = new Song("Mister Sandman");
Song song2 = new Song("Amazing Grace");
playList.add(song1);
playList.add(song2);
...
int numOfSongs = playList.size();
if (numOfSongs <= 24) {
    System.out.println("Do not book");
} else if ((numOfSongs > 24) & (numOfSongs < 50)) {
    System.out.println("Book for one night");
} else if ((numOfSongs >= 50)) {
    System.out.println("Book for two nights");
} else {
    System.out.println("Book for the week");
}
```

## Оператор `switch`

Оператор `switch` является управляющим, передавая управление одному из следующих за ним операторов `case` в зависимости от значения заданного в нем выражения. Оператор `switch` способен оперировать данными типа `char`, `byte`, `short`, `int`, оболочками типа `Character`, `Byte`, `Short` и `Integer`, перечислениями и символьными строками (в виде объектов типа `String`). Поддержка объектов `String` была внедрена только в версии `Java SE 7`. Оператор `break` служит для завершения работы оператора `switch`. Если в операторе `case` отсутствует оператор `break`, то выполняется следующая за ним строка кода (как правило, следующий оператор `case`).

И это продолжается до тех пор, пока не будет достигнут оператор `break` или конец оператора `switch`. Допускается единственная метка `default`, которая обычно указывается в конце оператора `switch` ради повышения удобочитаемости кода, как показано ниже.

```
String style;
String guitarist = "Eric Clapton";
...
switch (guitarist) {
    case "Chet Atkins":
        style = "Nashville sound";
        break;
    case "Thomas Emmanuel":
        style = "Complex fingerstyle";
        break;
    default:
        style = "Unknown";
        break;
}
```

## Итеративные операторы

Операторы простого и расширенного цикла `for`, а также операторы `while` и `do-while` являются итерационными. Они служат для циклического выполнения определенных фрагментов кода.

### Оператор цикла `for`

Оператор цикла `for` состоит из трех частей: инициализации, условного выражения и обновления. Как показано ниже, перед использованием этого оператора должна быть инициализирована переменная цикла (`i`). Условное выражение (`i < bArray.length()`) всегда вычисляется до начала выполнения цикла. Итерация (т.е. шаг цикла) начинается лишь в том случае, если результатом вычисления условного выражения оказывается истинное значение, а переменная цикла обновляется (`i++`) после каждой итерации.

```
Banjo [] bArray = new Banjo[2];
bArray[0] = new Banjo();
bArray[0].setManufacturer("Windsor");
bArray[1] = new Banjo();
bArray[1].setManufacturer("Gibson");
```



```
for (int i=0; i<bArray.length; i++){
    System.out.println(bArray[i].getManufacturer());
}
```

## Оператор расширенного цикла `for`

Оператор расширенного цикла `for`, т.е. цикла типа `for in` или `for each`, служит для циклической обработки объекта или массива, допускающего перебор его элементов. Цикл выполняется однократно для каждого элемента массива или коллекции без применения счетчика, поскольку количество итераций известно заранее.

```
ElectricGuitar eGuitar1 = new ElectricGuitar();
eGuitar1.setName("Blackie");
ElectricGuitar eGuitar2 = new ElectricGuitar();
eGuitar2.setName("Lucille");
ArrayList <ElectricGuitar> eList = new ArrayList<>();
eList.add(eGuitar1); eList.add(eGuitar2);
for (ElectricGuitar e : eList) {
    System.out.println("Name:" + e.getName());
}
```

## Оператор цикла `while`

В операторе цикла `while` сначала вычисляется условное выражение, и цикл выполняется лишь в том случае, если результатом вычисления условного выражения оказывается истинное значение. Выражение может быть типа `boolean` или `Boolean`.

```
int bandMembers = 5;
while (bandMembers > 3) {
    CoffeeShop c = new CoffeeShop();
    c.performGig(bandMembers);
    // задать произвольно от нуля до семи
    // членов музыкального коллектива
    bandMembers = new Random().nextInt(8);
}
```

## Оператор цикла `do-while`

Цикл в операторе `do-while` всегда выполняется хотя бы один раз. И его выполнение будет продолжаться до тех пор, пока результатом вычисления условного выражения не окажется истинное значение. Условное выражение может быть типа `boolean` или `Boolean`.

```
int bandMembers = 1;
do {
    CoffeeShop c = new CoffeeShop();
    c.performGig(bandMembers);
    Random generator = new Random();
    bandMembers = generator.nextInt(7) + 1; // 1-7
} while
```

## Передача управления

Операторы передачи управления служат для изменения хода выполнения программы. К их числу относятся операторы `break`, `continue` и `return`.

## Оператор `break`

Оператор `break` без метки служит для выхода из текущей ветви оператора `switch` или немедленного выхода из цикла. Этот оператор способен прервать работу простого и расширенного цикла `for`, а также циклов типа `while` и `do-while`.

```
Song song = new Song("Pink Panther");
Guitar guitar = new Guitar();
int measure = 1; int lastMeasure = 10;
while (measure <= lastMeasure) {
    if (guitar.checkForBrokenStrings()) {
        break;
    }
    song.playMeasure(measure);
    measure++;
}
```

А оператор `break` с меткой прерывает выполнение оператора цикла, следующего сразу за этой меткой. Метки обычно применяются во вложенных циклах `for` и `while`, когда требуется точно определить, из какого именно цикла следует выйти. Чтобы пометить какой-нибудь цикл или оператор, достаточно ввести оператор с меткой непосредственно перед помечаемым циклом или оператором, как показано в приведенном ниже примере кода.

```
...
playMeasures:
while (isWithinOperatingHours()) {
    while (measure <= lastMeasure) {
        if (guitar.checkForBrokenStrings()) {
            break playMeasures;
        }
        song.playMeasure(measure);
        measure++;
    }
} // выход из цикла здесь
```

## Оператор `continue`

Выполнение оператора `continue` без метки приводит к прекращению текущего шага простого или расширенного цикла `for`, а также цикла `while` или `do-while` и началу следующего шага соответствующего цикла. При этом будет выполнена проверка условий выполнения цикла. А оператор `continue` с меткой приводит к выполнению шага цикла, следующего сразу же после метки.

```
for (int i=0; i<25; i++) {
    if (playList.get(i).isPlayed()) {
        continue;
    } else {
        song.playAllMeasures();
    }
}
```

## Оператор `return`

Оператор `return` служит для выхода из метода и возврата значения, если в объявлении метода указано, что данный метод должен вернуть значение определенного типа.

```
private int numberOfFrets = 18; // по умолчанию
...
public int getNumberOfFrets() {
    return numberOfFrets;
}
```

Оператор `return` указывать необязательно, если он является последним в теле метода, который ничего не возвращает.

## Оператор `synchronized`

Ключевое слово `synchronized` может использоваться в Java для предоставления доступа к определенным участкам кода (например, к отдельным методам) только в одном потоке исполнения. Оператор `synchronized` позволяет управлять доступом к ресурсам, которые совместно используются в нескольких потоках исполнения. Подробнее об этом — в главе 14.

## Оператор `assert`

*Утверждения* являются логическими выражениями, которые используются в режиме отладки кода с целью проверить, ведет ли себя код именно так, как и ожидалось. Они активируются интерпретатором Java при запуске приложения и командной строки с параметром `-enableassertions` или `-ea`. Утверждения обозначаются следующим образом:

```
assert логическое_выражение;
```

Утверждения облегчают выявление ошибок, в том числе обнаружение неожиданных значений. Они предназначены для подтверждения предположений, которые всегда должны быть истинны (`true`). Если при выполнении прикладного кода

в режиме отладки утверждение окажется ложным (`false`), то генерируется исключение типа `java.lang.AssertionError` и происходит выход из программы. В противном случае ничего не происходит.

```
// значение переменной strings должно быть равно
// 4, 5, 6, 7, 8 или 12
assert (strings == 12 ||
        (strings >= 4 && strings <= 8));
```

Утверждения следует активизировать явным образом. Аргументы командной строки, предназначенные для активизации утверждений, описаны в главе 10.

В утверждении можно также указать необязательный код ошибки, как показано ниже. Несмотря на такое название, код ошибки на самом деле является обычным текстом или значением, используемым исключительно в информативных целях.

```
assert boolean_expression : errorcode;
```

Если утверждение, в котором содержится код ошибки, оказывается ложным, значение кода ошибки преобразуется в символьную строку и отображается для пользователя непосредственно перед выходом из программы. Ниже приведен пример утверждения, в котором используется код ошибки.

```
// показать неверное количество струн музыкального
// инструмента в переменной strings
assert (strings == 12 ||
        (strings >= 4 && strings <= 8)) :
        "Invalid string count: " + strings;
```

## Операторы обработки исключений

Операторы обработки исключений служат для указания кода, который должен быть выполнен в исключительных ситуациях. Для обработки исключений предназначены ключевые слова `throw` и `try/catch/finally`, обозначающие соответствующие операторы. Подробнее об обработке исключений речь пойдет в главе 7.