Глава 9

ИСКЛЮЧЕНИЯ И ОШИБКИ

Существуют только ошибки.

Аксиома Робертса

Что для одного ошибка, для другого — исходные данные.

Следствие Бермана из аксиомы Робертса

Никогда не выявляйте в программе ошибки, если не знаете, что с ними делать.

Руководство Штейнбаха

Иерархия исключений и ошибок

Исключительные ситуации (исключения) и ошибки возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Обычно считается, что исключения и ошибки — тождественные понятия. Примерами «популярных» ошибок являются: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на ноль. При возникновении исключения в приложении создается объект, описывающий это исключение. Затем текущий ход выполнения приложения останавливается, и включается механизм обработки исключений. При этом ссылка на объект-исключение передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы. Если в классе используется метод, в котором может возникнуть проверяемая исключительная ситуация, но не предусмотрена ее обработка, то ошибка возникает еще на этапе компиляции. При создании такого метода программист обязан включить в код метода обработку исключений, которые могут генерироваться в этом методе, или передать обработку исключения на более высокий уровень методу, вызвавшему данный метод. Схема обработки исключения подобна схеме обработки событий.

Исключение не должно восприниматься как нечто вредное, от которого следует избавиться любой ценой. Исключение — это источник дополнительной информации о ходе выполнения приложения. Такая информация позволяет лучше адаптировать код к конкретным условиям его использования, а также на ранней стадии выявить ошибки или защититься от их возникновения в будущем. В противном

случае «подавление» исключений приведет к тому, что о возникшей ошибке никто не узнает или узнает на стадии некорректно обработанной информации. Поиск места возникновения ошибки может быть затруднительным.

Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при ее появлении. Если подходящего класса не существует, то он может быть создан разработчиком. Все исключения являются наследниками суперкласса **Throwable** и его подклассов **Error** и **Exception** из пакета **java.lang**.

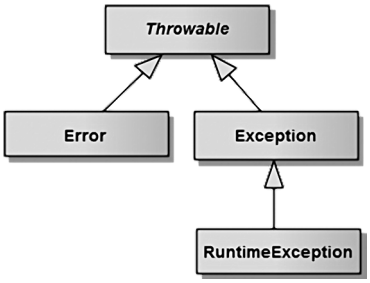

Рис. 9.1. *Иерархия основных классов исключений*

Исключительные ситуации типа **Error** возникают только во время выполнения программы. Такие исключения, связанные с серьезными ошибками, к примеру, с переполнением стека, не подлежат исправлению и не могут обрабатываться приложением. Собственные подклассы от **Error** создавать мало смысла по причине невозможности управления прерываниями. Некоторые классы из иерархии, наследуемых от класса **Error**, приведены на рисунке 9.2.

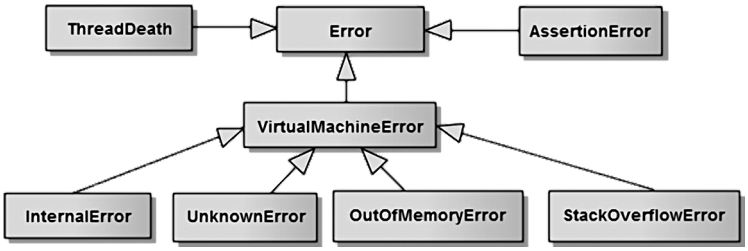

Рис. 9.2. *Некоторые классы исключений, наследуемые от класса Error*

На рисунке 9.3 приведена иерархия классов проверяемых исключений, наследуемых от класса **Exception** при отсутствии в цепочке наследования класса **RuntimeException**. Возможность возникновения проверяемого исключения может быть отслежена еще на этапе компиляции кода. Компилятор проверяет, может ли данный метод генерировать или обрабатывать исключение.

Проверяемые исключения должны быть обработаны в методе, который может их генерировать, или включены в **throws**-список метода для дальнейшей обработки в вызывающих методах.

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы. Список этих исключений приведен на рисунке 9.4. В отличие от проверяемых исключений, класс **RuntimeException** и порожденные от него классы относятся к непроверяемым

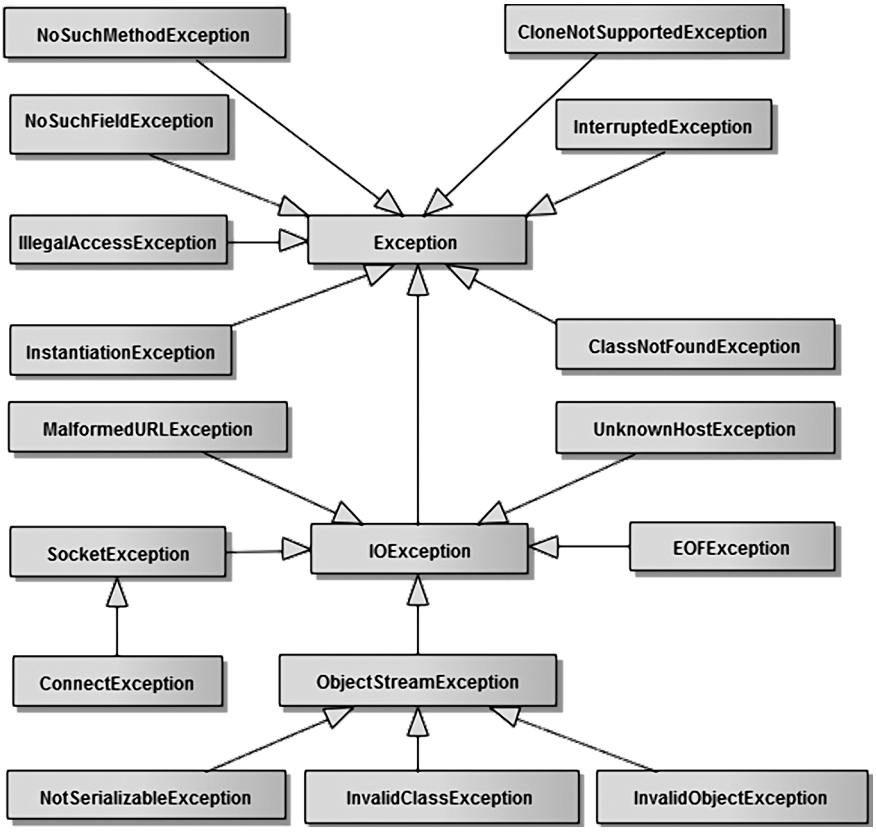

Рис. 9.3. *Иерархия классов, проверяемых (checked) исключительных ситуаций*

исключениям. Компилятор не проверяет, может ли генерировать и/или обрабатывать метод эти исключения. Исключения типа **RuntimeException** генерируются при возникновении ошибок во время выполнения приложения.

Ниже приведен список часто встречаемых в практике программирования непроверяемых исключений, знание причин возникновения которых необходимо при создании качественного кода.

Почему возникла необходимость деления исключений на проверяемые и непроверяемые? Представим, что следующие ситуации проверяются на этапе компиляции, а именно:

- деление в целочисленных типах вида **a/b** при **b=0** генерирует исключение **ArithmeticException**;
- индексация массивов, строк, коллекций. Выход за пределы такого объекта приводит к исключению **ArrayIndexOutOfBoundsException** и аналогичных;
- вызов метода на ссылке вида **obj.method()**, если **obj** ссылается на **null**.

Исключение	Значение
ArithmeticException	Арифметическая ошибка: деление на ноль и др.
ArrayIndexOutOfBoundsException	Индекс массива находится вне его границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
ConcurrentModificationException	Некорректный способ модификации коллекции
IllegalArgumentException	При вызове метода использован некорректный аргумент
IllegalMonitorStateException	Незаконная операция монитора на разблокированном объекте
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ коллекции
NegativeArraySizeException	Попытка создания массива с отрицательным размером
NullPointerException	Недопустимое использование ссылки на null
NumberFormatException	Невозможное преобразование строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексации вне границ строки
MissingResourceException	Отсутствие файла ресурсов properties или имени ресурса в нем
EnumConstantNotPresentException	Несуществующий элемент перечисления
UnsupportedOperationException	Встретилась неподдерживаемая операция

Рис. 9.4. *Классы непроверяемых исключений, наследуемых от класса RuntimeException*

Если бы возможность появления перечисленных исключений проверялась на этапе компиляции, то любая попытка индексации массива или каждый вызов метода требовали бы или блока **try-catch**, или секции **throws**. Такой код был бы практически непригоден для понимания и поддержки, поэтому часть исключений была выделена в группу непроверяемых и ответственность за защиту приложения от последствий их возникновения возложена на программиста.

Способы обработки исключений

Если при возникновении исключения в текущем методе обработчик не будет обнаружен, то его поиск будет продолжен в методе, вызвавшем данный метод, и так далее вплоть до метода **main()** для консольных приложений или другого метода, запускающего соответствующее приложение. Если же и там исключение не будет перехвачено, то JVM выполнит аварийную остановку приложения с вызовом метода **printStackTrace()**, выдающего данные трассировки.

Для проверяемого исключения возможность его генерации отслеживается. Передача обработки вызывающему методу осуществляется с помощью оператора

throws. В конце концов исключение может быть передано в метод **main()**, где и находится крайняя точка обработки. Добавлять оператор **throws** методу **main()** представляется дурным тоном программирования, как безответственное действие программиста, не обращающего никакого внимания на альтернативное выполнение программы.

На практике используется один из двух способов обработки исключений:

- перехват и обработка исключения в блоке **try-catch** метода;
- объявление исключения в секции **throws** метода и передача вызывающему методу (в первую очередь для проверяемых исключений).

Первый подход можно рассмотреть на следующем примере. При преобразовании содержимого строки к числу в определенных ситуациях может возникать проверяемое исключение типа **ParseException**. Например:

```
public double parseFromFrance(String numberStr) {
  NumberFormat format = NumberFormat.getInstance(Locale.FRANCE);
  double numFrance = 0;
  try {
    numFrance = format.parse(numberStr).doubleValue();
  } catch (ParseException e) { // checked exception
   // 1. throwing a standard exception,: IllegalArgumentException() – not very good
   // 2. throwing a custom exception, where ParseException as a parameter
   // 3. setting the default value - if possible
   // 4. logging if an exception is unlikely
  }
  return numFrance;
}
```

Исключительная ситуация возникнет в случае, если переданная строка содержит нечисловые символы или не является числом. Генерируется объект исключения, и управление передается соответствующему блоку **catch**, в котором он обрабатывается, иначе блок **catch** пропускается. Блок **try** похож на обычный логический блок. Блок **catch(){}** похож на метод, принимающий в качестве единственного параметра ссылку на объект-исключение и обрабатывающий этот объект.

Второй подход демонстрируется на этом же примере. Метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, вызывающим данный метод. В этом случае метод должен объявить о таком поведении с помощью ключевого слова **throws**, чтобы вызывающий метод мог защитить себя от этих исключений. В вызывающем методе должна быть предусмотрена или обработка этих исключений, или последующая передача соответствующему методу.

При этом объявляемый метод может содержать блоки **try-catch**, а может и не содержать их. Например, метод **parseFrance()** можно объявить:

```
public double parseFrance(String numberStr) throws ParseException {
  NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
  double numFrance  = formatFrance.parse(numberStr).doubleValue();
```

```
  return numFrance;
}
```

В практическом программировании такой подход допустим для **private**-методов.

Ключевое слово **throws** после имени метода позволяет разобраться с исключениями методов «чужих» классов, код которых отсутствует. Обрабатывать исключение при этом должен будет метод, вызывающий **parseFrance()**:

```
public void doAction() {
  // code here
  try {
    parseFrance(numberStr);
  } catch (ParseException e) {
    // code
  }
}
```

Создание и применение собственных исключений будет рассмотрено ниже в этой главе.

Обработка нескольких исключений

Если в блоке **try** может быть сгенерировано в разных участках кода несколько типов исключений, то необходимо наличие нескольких блоков **catch**, если только блок **catch** не обрабатывает все типы исключений.

/ # 1 # обработка двух типов исключений # */*

```
public void doAction() {
  try {
    int a = (int) (Math.random() * 2);
    System.out.println("a = " + a);
    int c[] = { 1 / a }; // place of occurrence of exception #1
    c[a] = 71; // place of occurrence of exception #2
  } catch (ArithmeticException e) {
    System.err.println("divide by zero " + e);
  } catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("out of bound: " + e);
  } // end try-catch block
  System.out.println("after try-catch");
}
```

Исключение **ArithmeticException** при делении на 0 возникнет при инициализации элемента массива **c[0]** действием **1/a** при **a=0**. В случае **a=1** генерируется исключение «превышение границ массива» при попытке присвоить значение второму элементу массива **c[]**, содержащего только один элемент. Однако пример, приведенный выше, носит чисто демонстративный характер

и не является образцом хорошего кода, так как в этой ситуации можно было обойтись простой проверкой аргументов на допустимые значения перед выполнением операций. К тому же генерация и обработка исключения — операция значительно более ресурсоемкая, чем вызов оператора **if** для проверки аргумента. Исключения должны применяться только для обработки исключительных ситуаций, и если существует возможность обойтись без них, то следует так и поступить.

Подклассы исключений в блоках **catch** должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения. Например:

```
try { //...
} catch(IllegalArgumentException e) { //...
} catch(PatternSyntaxException e) { // unreachable code
}
```

где класс **PatternSyntaxException** представляет собой подкласс класса **IllegalArgumentException**. Корректно будет просто поменять местами блоки **catch**:

```
try {
} catch(PatternSyntaxException e) { //..
} catch(IllegalArgumentException e) { //..
}
```

На практике иногда возникают ситуации, когда инструкций **catch** несколько, и обработка производится идентичная, например, вывод сообщения об исключении в журнал.

```
try {
  // some code
} catch(NumberFormatException e) {
  e.printStackTrace(); // or log
} catch(ClassNotFoundException e) {
  e.printStackTrace(); // or log
} catch(InstantiationException e) {
  e.printStackTrace(); // or log
}
```

В версии Java 7 появилась возможность объединить все идентичные инструкции в одну, используя для разделения оператор «|».

```
try {
  // some code
} catch(NumberFormatException | ClassNotFoundException | InstantiationException e){
  e.printStackTrace();
}
```

В **catch** не могут находиться исключения из одной иерархической цепочки. Такая запись позволяет избавиться от дублирования кода.

Введено понятие более точной переброски исключений (*more precise rethrow*). Это решение применимо в случае, если обработка возникающих исключений не предусматривается в методе и должна быть передана вызывающему данный метод методу.

До введения этого понятия код выглядел так:

```
public double parseFromFileBefore(String filename)
        throws FileNotFoundException, ParseException, IOException {
  NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
  double numFrance = 0;
  BufferedReader bufferedReader = null;
  try {
    FileReader reader = new FileReader(filename);
    bufferedReader = new BufferedReader(reader);
    String number = bufferedReader.readLine();
    numFrance = formatFrance.parse(number).doubleValue();
  } catch (FileNotFoundException e) {
    throw e;
  } catch (IOException e) {
    throw e;
  } catch (ParseException e) {
    throw e;
  } finally {
    if (bufferedReader != null) {
      bufferedReader.close();
    }
  }
  return numFrance;
}
```

More precise rethrow разрешает записать в единственную инструкцию **catch** более общее исключение, чем может быть генерировано в инструкции **try**, с последующей генерацией перехваченного исключения для его передачи в вызывающий метод.

```
public double parseFile(String filename)
        throws FileNotFoundException, ParseException, IOException {
  NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
  double numFrance = 0;
  BufferedReader bufferedReader = null;
  try {
    FileReader reader = new FileReader(filename);
    bufferedReader = new BufferedReader(reader);
    String number = bufferedReader.readLine();
    numFrance = formatFrance.parse(number).doubleValue();
  } catch (final Exception e) { // final – optional
    throw e; // more precise rethrow
  } finally {
    if (bufferedReader != null) {
```

```
      bufferedReader.close();
    }
  }
  return numFrance;
}
```

Наличие секции **throws** контролируется компилятором на предмет точного указания списка проверяемых исключений, которые могут быть генерированы в блоке **try-catch**. При возможности возникновения непроверяемых исключений последние в секции **throws** обычно не указываются. Ключевое слово **final** не позволяет подменить экземпляр исключения для передачи за пределы метода. Однако данную конструкцию можно использовать и без **final**.

Операторы **try** можно вкладывать друг в друга. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возникшему исключению, поиск будет развернут на одну ступень выше и будут проверены разделы **catch** внешнего оператора **try**.

/ # 2 # вложенные блоки try-catch # */*

```
try { // outer block
  int a = (int) (Math.random() * 2) - 1;
  System.out.println("a = " + a);
  try { // inner block
    int b = 1 / a;
    StringBuilder builder = new StringBuilder(a);
  } catch (NegativeArraySizeException e) {
    System.err.println("invalid buffer size: " + e);
  }
} catch (ArithmeticException e) {
  System.err.println("divide by zero: " + e);
}
```

В результате запуска приложения при **a=0** будет сгенерировано исключение **ArithmeticException**, а подходящий для его обработки блок **try-catch** является внешним по отношению к месту генерации исключения. Этот блок и будет задействован для обработки возникшей исключительной ситуации. Вкладывание блоков **try-catch** друг в друга загромождает код, поэтому такими конструкциями следует пользоваться с осторожностью.

Оператор throw

При разработке кода возникают ситуации, когда в приложении необходимо инициировать генерацию исключения для указания, например, на заведомо ошибочный результат выполнения операции, на некорректные значения параметра метода и др. Для генерации исключительной ситуации и создания экземпляра исключения используется оператор **throw**. В качестве исключения должен

быть использован объект подкласса класса **Throwable**, а также ссылки на них. Общая форма записи инструкции **throw**, генерирующей исключение:

```
throw subclassThrowable;
```

Объект-исключение может уже существовать или создаваться с помощью оператора **new**:

```
throw new IllegalArgumentException();
```

При достижении оператора **throw**, выполнение кода прекращается. Ближайший блок **try** проверяется на наличие соответствующего обработчика **catch**. Если он существует, управление передается ему, иначе проверяется следующий из вложенных операторов **try**. Инициализация объекта-исключения без оператора **throw** никакой исключительной ситуации не вызовет.

В ситуации, когда получение методом достоверной информации критично для выполнения им своей функциональности, у программиста может возникнуть необходимость в генерации исключения, так как метод не может выполнить ожидаемых от него действий, основываясь на некорректных или ошибочных данных. Ниже приведен пример, в котором оператор **throw** генерирует исключение, обрабатываемое виртуальной машиной при выбросе из метода **main()**.

/ # 3 # генерация исключений # ActionMain.java # ResourceAction.java # Resource.java */*

```
package by.epam.learn.exception;
public class ResourceAction {
  public static void load(Resource resource) {
    if (resource == null || !resource.exists() || !resource.isCreate()) {
      throw new IllegalArgumentException();
      // better custom exception, eg., throw new ResourceException();
    }
    // more code
  }
}
package by.epam.learn.exception;
public class ActionMain {
  public static void main(String[] args) {
    Resource resource = new Resource(); //or Resource resource = null;//!!
    ResourceAction.load(resource);
  }
}
package by.epam.learn.exception;
public class Resource {
  // fields
  public boolean isCreate() {
    // more code
  }
```

```
  public boolean exists() {
    // more code
  }
  public void execute() {
    // more code
  }
  public void close() {
    // more code
  }
}
```

Вызываемый метод **load()** может при отсутствии требуемого ресурса или при аргументе **null** генерировать исключение, перехватываемое обработчиком. В результате экземпляр непроверяемого исключения **IllegalArgumentException** как подкласса класса **RuntimeException** передается обработчику исключений в методе **main()**.

Собственные исключения

Для повышения качества и скорости восприятия кода разработчику следует создать собственное исключение как подкласс класса **Exception**, а затем использовать его при обработке ситуации, не являющейся исключением с точки зрения языка, но нарушающей логику вещей. По соглашению наследник любого класса-исключения должен заканчиваться словом **Exception**.

В случае генерации собственного проверяемого исключения **ResourceException**, компилятор требует обработки объекта исключения в методе или передачи его с помощью инструкции **throws**. Собственное исключение может быть создано как

/ # 4 # класс собственного исключения # ResourceException.java */*

```
public class ResourceException extends Exception {
  public ResourceException() {
  }
  public ResourceException(String message, Throwable cause) {
    super(message, cause);
  }
  public ResourceException(String message) {
    super(message);
  }
  public ResourceException(Throwable cause) {
    super(cause);
  }
}
```

У подкласса класса **Exception** обычно определяются минимум три конструктора, два из которых в качестве параметра принимают объект типа **Throwable**, что означает генерацию исключения на основе другого исключения.

Один из параметров — сообщение, которое может быть выведено в поток ошибок; другой — реальное исключение, которое привело к вызову собственного исключения. Этот подход показывает, как можно сохранить дополнительную информацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину генерации исключения, он всего лишь должен вызвать метод **getCause()**. Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки собственного исключения. Иногда цепочка вложенных исключений может быть достаточно большой, но всегда остается возможность на любом этапе узнать первопричину возникновения цепочки. Тогда метод **load()** с применением собственного исключения **ResourceException** для класса **ResourceAction** будет выглядеть так:

```
public static void load(Resource resource) throws ResourceException {
  if (resource == null || !resource.exists() || !resource.isCreate()) {
    throw new ResourceException();
  }
  // more code
}
```

В этом случае в вызывающем методе обработка исключения будет обязательна.

Если же генерируется стандартное исключение или получены такие значения некоторых параметров, что генерация какого-либо стандартного исключения становится неизбежной немедленно либо сразу по выходе из метода, то следует генерировать собственное исключение.

```
public int loadFile(String filename) throws ResourceException {
  int data;
  try {
    FileReader reader = new FileReader(filename);
    data = reader.read();
  } catch (IOException e) {
    throw new ResourceException(e);
  }
  return data;
}
```

Если метод генерирует исключение с помощью оператора **throw** и при этом блок **catch** в методе отсутствует, то для передачи обработки исключения вызывающему методу тип проверяемого (*checked*) класса исключений должен быть указан в операторе **throws** при объявлении метода.

```
Resource resource = new Resource();
  try {// required !!
    ResourceAction.load(resource);
  } catch(ResourceException e) {
    System.err.print(e);
}
```

Обязательно передавать перехваченное блоком **catch** исключение в качестве параметра для возможности развертывания стека исключений в момент его обработки.

Если ошибка некритическая, то вместо генерации исключения можно просто записать лог.

Разработчики программного обеспечения стремятся к высокому уровню повторного использования кода, поэтому они постарались предусмотреть и закодировать все возможные исключительные ситуации. При реальном программировании создание собственных классов исключений позволяет разработчику выделить важные аспекты приложения и обратить внимание на детали разработки.

Генерация непроверяемых исключений

Если без каких-то данных приложение не сможет нормально функционировать, значит, есть очень серьезная ошибка, поэтому приложение разумнее остановить, устранить ошибки и перезапустить. Например, файл содержит данные конфигурации пула соединений с СУБД, а по заданному пути файл отсутствует. Обработать такую ошибку в коде приложения невозможно:

```
public void loadFile(String filename) {
  try {
    FileReader reader = new FileReader(filename);
    // more code....
  } catch (FileNotFoundException e) {
    logger.fatal("fatal error: config file not found: " + filename, e);
    throw new RuntimeException("fatal: config file not found: " + filename, e);
  }
}
```

Это единственная разумная ситуация, когда допустимо генерировать непроверяемое исключение. Одновременная запись лога также допустима, чтобы была возможность зафиксировать сообщение о деталях ошибки.

Для исключений, являющихся подклассами класса **RuntimeException** (*unchecked*) и используемых для отображения программных ошибок, при выполнении приложения в объявлении метода секция **throws** может отсутствовать, так как играет только информационную роль.

В секции **throws** можно использовать стандартные исключения только в случае, если этот метод **private**. Такой подход упрощает восприятие кода и не нарушает правила генерации только собственных исключений, так как метод **private** вызывается только методами того же класса и не виден для других классов.

Блок finally

Возможна ситуация, при которой нужно выполнить некоторые действия по завершении программы (закрыть поток, освободить соединение с базой данных) вне зависимости от того, произошло исключение или нет. В этом случае используется блок **finally**, который обязательно выполняется после или инструкции **try**, или **catch**. Например:

```
try {
  // code
} catch(OneException e) {
  // code // not required
} catch(TwoException e) {
  // code // not required
} finally {
  // executed after try or after catch */
}
```

Каждому разделу **try** должен соответствовать по крайней мере один раздел **catch** или блок **finally**. Блок **finally** часто используется для закрытия файлов и освобождения других ресурсов, захваченных для временного использования в начале выполнения метода. Код блока выполняется перед выходом из метода даже в том случае, если перед ним были выполнены такие инструкции, как **throws**, **return**, **break**, **continue**.

/ # 5 # выполнение блоков finally # ResourceAction.java */*

```
package by.epam.learn.exception;
public class ResourceAction {
  public void doAction() {
    Resource resource = null;
    try {
      // init resources
      resource = new Resource(); // exception generation possible
      // using resources
      resource.execute(); // exception generation possible
      // resource.close(); // release of resources (incorrect)
    } catch (ResourceException e) {
      // code
    } finally {
      // release of resources (correct)
      if (resource != null) {
        resource.close();
      }
    }
    System.out.print("after finally");
  }
}
```

В методе **doAction()** при использовании ресурсов и генерации исключения осуществляется преждевременный выход из блока **try** с игнорированием всего оставшегося в нем кода, но до выхода из метода обязательно будет выполнен раздел **finally**. Освобождение ресурсов в этом случае произойдет корректно. В следующей главе будет рассмотрен новый способ закрытия ресурсов *autocloseable*.

Наследование и исключения

Создание сложных распределенных систем редко обходится без наследования и обработки исключений. Следует знать два правила для проверяемых исключений при наследовании:

- переопределяемый метод в подклассе не может содержать в инструкции **throws** исключений, не обрабатываемых в соответствующем методе суперкласса;
- конструктор подкласса должен включить в свою секцию **throws** все классы исключений или их суперклассы из секции **throws** конструктора суперкласса, к которому он обращается при создании объекта.

Первое правило имеет непосредственное отношение к расширяемости приложения. Пусть при добавлении в цепочку наследования нового класса его полиморфный метод включил в блок **throws** «новое» проверяемое исключение из другой цепочки наследования или исключение суперкласса текущего. Тогда методы логики приложения, принимающие объект нового класса в качестве параметра и вызывающие данный полиморфный метод, не готовы обрабатывать «новое» исключение, так как ранее в этом не было необходимости. Поэтому при попытке добавления «нового» *checked*-исключения в секцию **throws** полиморфного метода компилятор выдает сообщение об ошибке.

/ # 6 # полиморфизм и исключения # */*

```
package by.epam.learn.exception;
public class Stone {
  public void accept(String data) throws ResourceException {
    /* more code */
  }
package by.epam.learn.exception;
public class GreenStone extends Stone {
  @Override
  public void accept(String data)  {
    //some code
  }
}
package by.epam.learn.exception;
public class WhiteStone extends Stone {
  @Override
  public void accept(String data) throws ResourceException {
```

```
    super.accept(data);
  }
}
package by.epam.learn.exception;
import java.io.FileWriter;
import java.io.IOException;
public class GreyStone extends Stone {
  @Override
  public void accept(String data) throws IOException {//compile error
    FileWriter writer = new FileWriter("data.txt");
  }
}
package by.epam.learn.exception;
public class StoneService {
  public void buildHouse(Stone stone) {
    try {
      stone.accept("some info");
    } catch(ResourceException e) {
      // handling of ResourceException and its subclasses
      System.err.print(e);
    }
  }
}
```

Если при объявлении метода суперкласса инструкция **throws** присутствует, то в подклассе эта инструкция может вообще отсутствовать или в ней могут быть объявлены только исключения, являющиеся подклассами исключения из секции **throws** метода суперкласса. Второе правило позволяет защитить программиста от возникновения неизвестных ему исключений при создании объекта.

/ # 7 # конструкторы и исключения # Resource.java # ConcreteResource.java # NonConcreteResource.java */*

```
import java.io.FileNotFoundException;
import java.io.IOException;
class Resource { // old class
  public Resource(String filename) throws FileNotFoundException {
    // more code
  }
}
class ConcreteResource extends Resource { // old class
  public ConcreteResource(String name) throws FileNotFoundException {
    super(name);
    // more code
  }
  public ConcreteResource() throws IOException {
    super("file.txt");
    // more code
  }
}
```

```
class NonConcreteResource extends Resource { // new class
  public NonConcreteResource(String name, int mode) { /* compile error */
    super(name);
    // more code
  }
  public NonConcreteResource(String name, int mode, String type)
                      throws ParseException { /* compile error */
    super(name);
    // more code
  }
}
```

Если разрешить создание экземпляра в виде:

```
NonConcreteResource inCorrect = new NonConcreteResource("info", 1);
```

то конструктор суперкласса может сгенерировать исключение и никаких предварительных действий по его предотвращению принято не будет.

```
try {
    NonConcreteResource correct = new NonConcreteResource();
} catch(ParseException e) {
    // trace
}
```

В приведенном выше случае компилятор не разрешит создать конструктор подкласса, обращающийся к конструктору суперкласса без корректной инструкции **throws**. Если бы это было возможно, то при создании объекта подкласса класса **NonConcreteResource** не было бы никаких сообщений о возможности генерации исключения, и при возникновении исключительной ситуации ее источник было бы трудно идентифицировать.

Ошибки статической инициализации

Какое значение получит статическая переменная класса, если при ее инициализации будет сгенерировано исключение? Подразумевается, что сразу после инициализации переменной можно воспользоваться. А как это сделать, если инициализация не прошла? Действительно, такой переменной воспользоваться не получится, и все исключения, возникающие в процессе инициализации, получают обертку в виде **ExceptionInInitializerError**, которая останавливает JVM, предоставляя программисту возможность устранить первопричину ошибки непосредственной коррекцией кода или ресурсов приложения.

/ # 8 # генерация исключения в статическом поле # IdGenerator.java*

```
package by.epam.learn.exception;
public class IdGenerator {
    final static int START_COUNTER;
```

```
    static {
       START_COUNTER = Integer.parseInt("Y-");
    }
}
```

и попытка запуска кода генерации исключения вида:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
          at by.bsu.exception.Runner.main(Runner.java:10)
          at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.
NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) sun.reflect.
DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.
Method.invoke(Method.java:483)
          at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
Caused by: java.lang.NumberFormatException: For input string: "Y-"
          at ...
```

При попытке в статическом блоке преобразовать недопустимое значение, в число будет сгенерировано исключение **NumberFormatException**, но JVM его обернет в ошибку статической инициализации **ExceptionInInitializerError**. Константа **START_COUNTER** может быть инициализирована только один раз, и второй попытки в данном запуске приложения уже быть не может, и использовать такую переменную не получится, поэтому и генерируется в итоге ошибка, а не исключение. Для инициализации данного поля требуется перезапуск приложения. Та же ошибка будет выброшена и при прямой инициализации статической переменной:

```
static int startCounter = Integer.parseInt("Y-");
```

На практике такие ситуации встречаются достаточно часто, когда при старте приложения необходимо считать конфигурационную информацию, например, содержащую данные для соединения с СУБД:

/ # 9 # генерация исключения в статическом поле # DbConfigManager.java */*

```
import java.util.ResourceBundle;
public class DbConfigManager {
    static ResourceBundle bundle = ResourceBundle.getBundle("resources.database");
    static String driverName = bundle.getString("driver.name");
}
```

Ошибка статической инициализации **ExceptionInInitializerError** возникнет как в случае отсутствия файла с указанным именем, так и в случае отсутствия в файле ключа **driver.name**. Только в этом случае основанием для генерации ошибки будет **java.util.MissingResourceException**.

Рекомендации по обработке исключений

Генерация исключения — процесс ресурсоемкий, и слишком частая генерация исключений оказывает влияние на быстродействие. Если генерация

исключения необходима, то следует воспользоваться следующими рекомендациями.

- Не обрабатывать конкретное исключение или несколько исключений с использованием в блоке **catch** исключения более общего типа.

```
try {
    // more code here
} catch (Exception e) { // or Throwable
    System.err.println(e);
}
```

Вместо этого следует классифицировать исключения:

```
try {
    // more code here
} catch (NegativeArraySizeException e) {
    System.err.println("invalid buffer size: " + e);
} catch (NumberFormatException e) {
    System.err.println("invalid character in number: " + e);
}
```

- Не оставлять пустыми блоки **catch**. При таком перехвате исключения пользователь не узнает, что исключительная ситуация имела место, и не станет устранять ее причины.

```
try {
    // some code here
} catch (NumberFormatException e) {
}
```

- По возможности не использовать одинаковую обработку различных исключений.

```
try {
    // some code here
} catch (IOException e) {
    e.printStackTrace();
} catch (SAXException e) {
    e.printStackTrace();
}
```

Для замены можно воспользоваться конструкцией *multi-catch* из Java 7 или в каждый блок **catch** помещать уникальную обработку.

- Не создавать класс исключений, эквивалентный по смыслу уже существующему. Такие исключения способны запутать разработчика, например:

```
public class NullPointException extends Exception {}
```

Исключение похоже на стандартное **NullPointerException** и может обмануть пользователя визуально. Или другой пример:

```
public class EnumNotPresentException extends Exception {}
```

вместо которого следует применить класс **EnumConstantNotPresentException**.

Прежде чем написать свое исключение, необходимо изучить документацию по стандартным исключениям, возможно, там найдется подходящее по смыслу. В обоих случаях непроверяемое исключение было подменено по смыслу на проверяемое, что недопустимо.

- Не создавать избыточное число классов собственных исключений. Прежде чем создавать новый класс исключений, следует подумать, что, возможно, ранее созданный в состоянии его обработать.
- Не использовать исключения в ситуациях, которые могут ввести в заблуждение:

```
public class Coin {
  private int weight;
  public void setWeight(int weight) throws IOException {
    if (weight <= 0) {
      throw new IOException();
    }
    this.weight = weight;
  }
}
```

- Не допускать, чтобы часть обработки ошибки присутствовала в блоке кода, генерирующем исключение:

```
public void setDeduce(double deduce) throws TaxException {
  if (deduce < 0) {
    this.deduce = 0; // unnecessary
    recalculateAmount(); // unnecessary
    System.err.print(DEDUCE_NEGATIVE); // unnecessary
    throw new TaxException("VAT deduce < 0");
  }
  this.deduce = deduce;
  recalculateAmount();
}
```

- Не допускать одновременной записи лога и генерации исключения, кроме непроверяемых исключений, в этом случае запись лога обязательна:

```
public void setDeduce(double deduce) throws TaxException {
  if (deduce < 0) {
    logger.error("VAT deduce is negative");
    throw new TaxException("VAT deduce is negative");
  }
  this.deduce = deduce;
  recalculateAmount();
}
```

- Никогда самостоятельно не генерировать **NullPointerException**, избегать случаев, когда такая генерация возможна в принципе. Проверка значения

ссылки на **null** позволяет обойтись без генерации исключения. Если по логике приложения необходимо генерировать исключение, следует использовать, например, **IllegalArgumentException** с соответствующей информацией об ошибке или собственное исключение.

- Никогда самостоятельно не перехватывать **NullPointerException** в блоке **catch**. Избежать этого можно простой проверкой значения ссылки на **null**. Желательно обходиться без перехватов и других непроверяемых исключений.
- Не следует в общем случае в секцию **throws** помещать *unchecked*-исключения.
- В любом случае, если есть возможность не генерировать исключение, следует ею воспользоваться.
- Не рекомендуется вкладывать блоки **try-catch** друг в друга из-за ухудшения читаемости кода.
- При создании собственных исключений следует проводить наследование от класса **Exception** либо от другого проверяемого класса исключений, а не от **RuntimeException**.
- Никогда не генерировать исключения в инструкции **finally**:

```
try {
  // possible throw exception
} finally {
  if (boolean_expression) {
    throw new CustomException();
  }
}
```

При такой генерации исключения никто в приложении не узнает об исключении и, соответственно, не сможет обработать исключение, ранее сгенерированное в блоке **try**, в случае, если оно не было обработано в блоке **catch**. В связи со сказанным никогда не следует использовать в блоке **finally** операторы **return**, **break**, **continue**.

- Во избежание дублирования логов, не следует писать логи из конструкторов классов-исключений:

```
public class CoinTechnicalException extends Exception {
  static Logger logger = LogManager.getLogger();
  public CoinTechnicalException() {
    logger.error(this.printStackTrace());
  }
  public CoinTechnicalException(String message, Throwable cause) {
    super(message, cause);
    logger.error(message, cause);
  }
}
```

Отладочный механизм assertion

Борьба за качество программ ведется различными способами. На этапе отладки найти неявные ошибки в функционировании приложения бывает довольно сложно. Механизм *assertion* позволяет проверять предположения о значениях данных в методах приложения. Если механизм отладки включен, то при нахождении некорректных данных генерируется **AssertionError**, с указанием места их обнаружения. Например, в методе, использующем линейные размеры какой-либо сущности, информация о размере извлекается из файла, и в результате может быть получено отрицательное значение. Далее неверные данные влияют на результат вычисления метода и т.д. Определять такие ситуации позволяет механизм проверочных утверждений (*assertion*). При помощи этого механизма можно сформулировать требования к входным, выходным и промежуточным данным непубличных методов классов в виде некоторых логических условий.

Стандартная попытка обработать ситуацию появления отрицательного размера может выглядеть следующим образом:

```
int size = pool.getPoolSize();
if (size > 0) {
    // more code
} else {
    // fatal error
}
```

Механизм *assertion* позволяет создать код, который будет генерировать **AssertionError** на этапе его отладки в результате проверки постусловия или промежуточных данных в виде:

```
int size = poll.getPoolSize();
assert (size > 0) : "incorrect PoolSize= " + size;
// more code
```

Правописание инструкции **assert**:

```
assert bool_exp : expression;
assert bool_exp;
```

Выражение **bool_exp** может принимать только значение типов **boolean** или **Boolean**, а **expression** — любое значение, которое может быть преобразовано к строке. Если логическое выражение получает значение **false**, то генерируется исключение **AssertionError** и выполнение программы прекращается с выводом на консоль значения выражения **expression** (если оно задано).

Механизм *assertion* хорошо подходит для проверки инвариантов, например, перечислений:

```
enum Mono { WHITE, BLACK }
String str = "WHITE"; // "GRAY"
Mono mono = Mono.valueOf(str);
```

```
// more code
switch (mono) {
    case WHITE : // more code
        break;
    case BLACK : // more code
        break;
    default :
        assert false : "Colored!";
}
```

Создатели языка не рекомендуют использовать *assertion* при проверке параметров **public**-методов. В таких ситуациях лучше рассматривать возможность генерации исключения одного из типов: **IllegalArgumentException** или собственное исключение. Нет также особого смысла в механизме *assertion* при проверке пограничных значений переменных, поскольку исключительные ситуации генерируются в этом случае без посторонней помощи.

Механизм *assertion* можно включать для отдельных классов или пакетов при запуске виртуальной машины в виде:

```
java -enableassertions RunnerMain
```

или

```
java -ea RunnerMain
```

Для выключения *assertion* применяется **-disableassertions** или **-da**.

Вопросы к главе 9

1. Что для программы является исключительной ситуацией? Какие существуют способы обработки ошибок в программах?
2. Что такое исключение для Java-программы? Что значит «программа генерировала\выбросила исключение»? Привести пример, когда исключения генерируются виртуальной машиной (автоматически) и когда необходимо их генерировать вручную.
3. Привести иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?
4. Объяснить работу оператора **try-catch-finally**. Когда данный оператор следует использовать? Сколько блоков **catch** может соответствовать одному блоку **try**?
5. Можно ли вкладывать блоки **try** друг в друга, можно ли вложить блок **try** в **catch** или **finally**? Как происходит обработка исключений, выброшенных внутренним блоком **try**, если среди его блоков **catch** нет подходящего?
6. Что называют стеком операторов **try**? Как работает блок **try** с ресурсами?
7. Указать правило расположения блоков **catch** в зависимости от типов перехватываемых исключений. Может ли перехваченное исключение быть

сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок **catch** выбрасывать иные исключения, и если да, то привести пример, когда это может быть необходимо.

8. Когда происходит вызов блока **finally**? Существуют ли ситуации, когда блок **finally** не будет вызван? Может ли блок **finally** выбрасывать исключения? Может ли блок **finally** выполниться дважды?
9. Как генерировать исключение вручную? Объекты каких классов могут быть генерированы в качестве исключений? Можно ли генерировать два исключения одновременно?
10. Объяснить, как работают операторы **throw** и **throws**. В чем их отличия?
11. Объяснить правила реализации секции **throws** при переопределении метода и при описании конструкторов производного класса.
12. Как ведет себя блок **throws** при работе с проверяемыми и непроверяемыми исключениями?
13. Каков будет результат создания объекта, если конструктор при работе сгенерирует исключительную ситуацию?
14. Нужно ли генерировать исключения, входящие в Java SE? Как создать собственные классы исключений?

Задания к главе 9

Вариант А

В символьном файле находится информация об N числах с плавающей запятой с указанием локали каждого числа отдельно. Прочитать информацию из файла. Проверить на корректность, то есть являются ли числа числами. Преобразовать к числовым значениям и вычислить сумму и среднее значение прочитанных чисел.

Создать собственный класс исключения. Предусмотреть обработку исключений, возникающих при нехватке памяти, отсутствии самого файла по заданному адресу, отсутствии или некорректности требуемой записи в файле, недопустимом значении числа (выходящим за пределы максимально допустимых значений) и т.д.

Тестовые задания к главе 9

Вопрос 9.1.

Дан код:

```
class Quest {
  static void method() throws ArithmeticException {
    int i = 7 / 0;
```