Глава 8
ИСКЛЮЧЕНИЯ И ОШИБКИ

Иерархия и способы обработки

Исключительные ситуации (исключения) возникают во время выполнения программы, когда возникшая проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Примерами являются особо «популярные»: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на нуль. При возникновении исключения 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 должен включить в код обработку исключений, которые могут генерировать этот метод, или передать обработку исключения на более высокий уровень методу, вызвавшему данный метод.

Каждой исключительной ситуации поставлен в соответствие некоторый класс. 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`**.

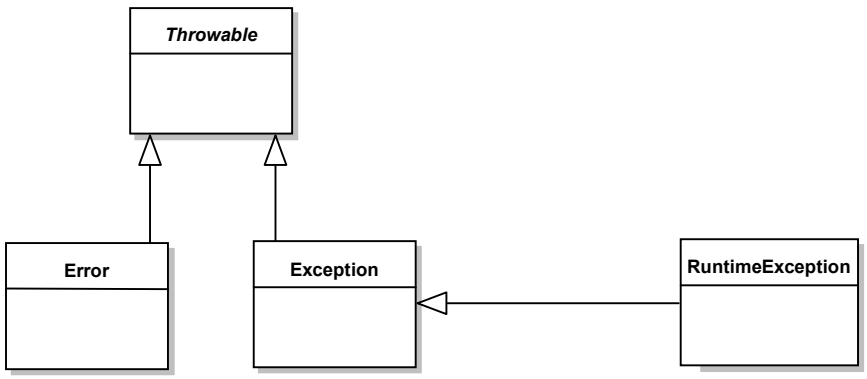

Рис. 8.1. Иерархия основных классов исключений

Исключительные ситуации типа **`Error`** возникают только во время выполнения программы. Такие исключения связаны с серьезными ошибками, к примеру – переполнение стека, и не подлежат исправлению и не могут обрабатываться приложением. Иерархия классов, наследуемых от класса **`Error`**, приведена на рисунке 8.2.

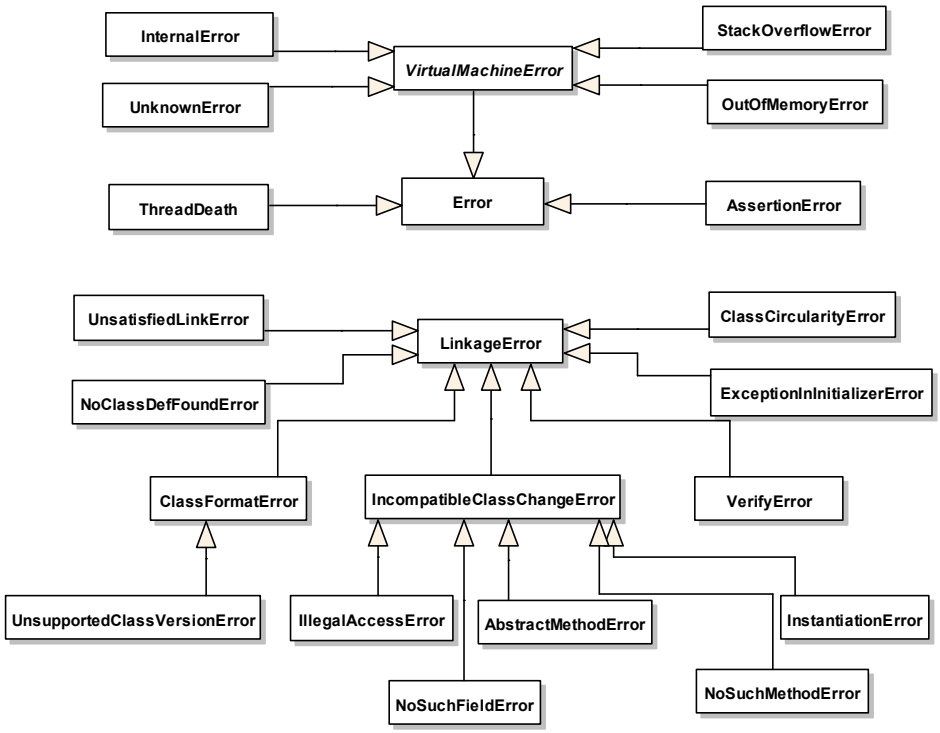

Рис. 8.2. Иерархия классов исключений, наследуемых от класса Error.

На рисунке 8.3 приведена иерархия классов исключений, наследуемых от класса **Exception**.

Проверяемые исключения должны быть обработаны в методе, который может их генерировать, или включены в **throws**-список метода для дальнейшей обработки в вызывающих методах. Возможность возникновения проверяемого исключения может быть отслежена на этапе компиляции кода.

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы. Иерархия этих исключений приведена на рисунке 8.4. В отличие от проверяемых исключений, класс **RuntimeException** и порожденные от него классы относятся к непроверяемым исключениям. Компилятор не проверяет, генерирует ли и обрабатывает ли метод эти исключения. Исключения типа **RuntimeException** автоматически генерируются при возникновении ошибок во время выполнения приложения. Таким образом, нет необходимости в проверке генерации исключения вида:

```
if(a==null) throw new NullPionterException();
```

объект класса **NullPionterException** при возникновении ошибки будет сгенерирован автоматически. Кроме этого, в любом случае нет необходимости в обработке этого исключения непосредственно в методе или в передаче на обработку вызывающему методу с помощью оператора **throw**. В конце концов исключение будет передано в метод **main()**, где обрабатывается вызовом метода **printStackTrace()**, выдающего данные трассировки.

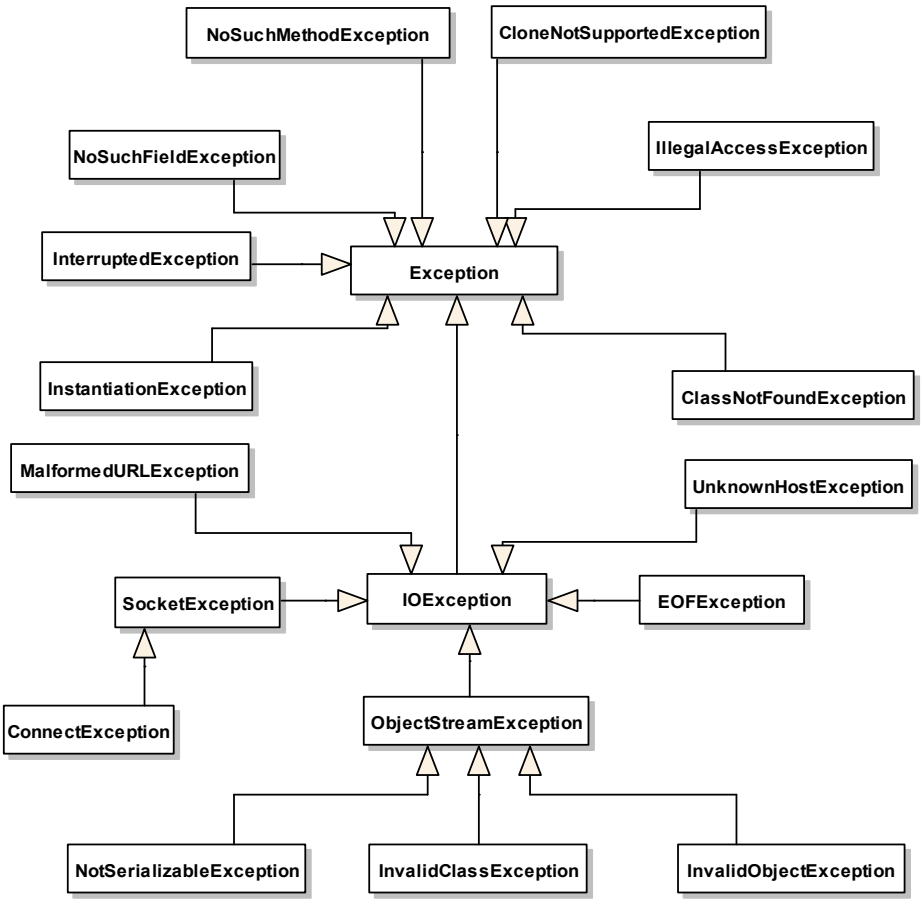

Рис. 8.3. Иерархия классов проверяемых (checked) исключительных ситуаций

Обычно используется один из трех способов обработки исключений:

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

Первый подход можно рассмотреть на следующем примере. При клонировании объекта в определенных ситуациях может возникать исключительная ситуация типа **`CloneNotSupportedException`**. Например:

```
public void changeObject(Student ob) {
            try {
                  Object temp = ob.clone();
                  //реализация
            } catch (CloneNotSupportedException e) {
                  System.err.print(e);
            }
      }
```

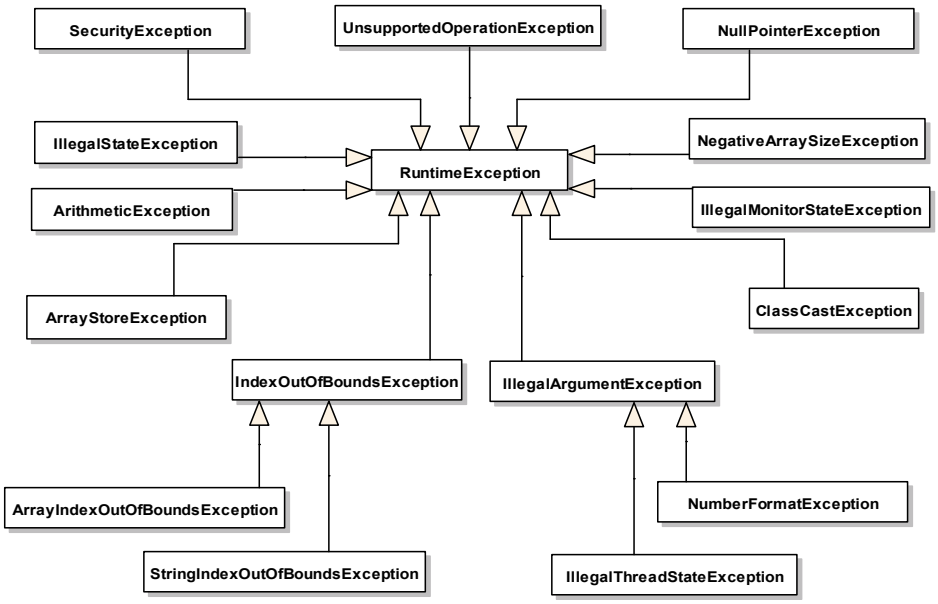

Рис. 8.4. Иерархия непроверяемых (unchecked) исключений

При клонировании может возникнуть исключительная ситуация в случае, если переданный объект не поддерживает клонирование (не включен интерфейс **`Cloneable`**). В этом случае генерируется соответствующий объект, и управление передается блоку **`catch`**, в котором обрабатывается данный тип исключения, 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 вызывающие методы могли защитить себя от этих исключений. В вызывающих методах должна быть предусмотрена обработка этих исключений. Форма объявления такого метода:

```
Тип имяМетода(список_аргументов)
        throws список_исключений { }
```

При этом сам таким образом объявляемый метод может содержать блоки **`try-catch`**, а может и не содержать их. Например, метод **`changeObject()`** можно объявить:

```
public void changeObject(Student ob)
            throws CloneNotSupportedException {
            Object temp = ob.clone();
                  //реализация
            }
```

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

```
public void load(Student stud) {
     try {
          changeObject(stud);
     } catch (CloneNotSupportedException e) {
          String error = e.toString();
          System.err.println(error);
     }
}
```

Создание собственных исключений будет рассмотрено позже в этой главе.

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.

```
/* пример # 1 : обработка двух типов исключений: TwoException.java */
package chapt08;

public class TwoException {
     public static void main(String[] args) {
          try {
               int a = (int)(Math.random() * 2);
               System.out.println("a = " + a);
               int c[] = { 1/a };
               c[a] = 71;
          } catch(ArithmeticException e) {
               System.err.println("деление на 0" + e);
          } catch(ArrayIndexOutOfBoundsException e) {
               System.err.println(
                       "превышение границ массива: " + e);
          }//последний catch
               System.out.println("после блока try-catch");
     }
}
```

Исключение "деление на 0" возникнет при инициализации элемента массива **a=0**. В противном случае (при **a=1**) генерируется исключение "превышение границ массива" при попытке присвоить значение второму элементу массива **c[]**, который содержит 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 {/*код, который может вызвать исключение*/
} catch(RuntimeException e) { /* суперкласс RuntimeException
                                 перехватит объекты всех своих подклассов */
```

```
} catch(ArithmeticException e) {} /* не может быть вызван,
                                     поэтому возникает ошибка компиляции */
```

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**.

```
/* пример # 2 : вложенные блоки try-catch: MultiTryCatch.java */
package chapt08;

public class MultiTryCatch {
     public static void main(String[] args) {
          try { // внешний блок
               int a = (int) (Math.random() * 2) - 1;
               System.out.println("a = " + a);
               try { // внутренний блок
                    int b = 1/a;
                    StringBuffer sb = new StringBuffer(a);
               } catch (NegativeArraySizeException e) {
                    System.err.println(
                         "недопустимый размер буфера: " + e);
               }
          } catch (ArithmeticException e) {
               System.err.println("деление на 0" + 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.

Третий подход к обработке исключений будет рассмотрен ниже на примере создания пользовательских исключений.

Оператор throw

В программировании часто возникают ситуации, когда программисту необходимо самому инициировать генерацию исключения для указания, например, на заведомо ошибочный результат выполнения операции, на некорректные значения параметра метода и др. Исключительную ситуацию можно создать с помощью оператора **throw**, если объект-исключение уже существует, или инициализировать его прямо после этого оператора. Оператор **throw** используется для генерации исключения. Для этого может быть использован объект класса **Throwable** или объект его подкласса, а также ссылки на них. Общая форма записи инструкции **throw**, генерирующей исключение:

```
throw объектThrowable;
```

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

```
throw new IOException();
```

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.

Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор **throw** генерирует исключение, обрабатываемое в разделе **catch**, в котором генерируется другое исключение.

```
/* пример # 3 : генерация исключений : ThrowGeneration.java */
package chapt08;
import java.io.File;

public class ThrowGeneration {
      public static void connectFile(File file) {
        if (file == null || !file.exists())
            throw new IllegalArgumentException();/*генерация
                                                  исключения */
            //...
       }
     public static void main(String[] args) {
          File f = new File("demo.txt");
          // File f = null;
          try {
               connectFile(f);
          } catch(IllegalArgumentException e) {
               System.err.print("обработка unchecked-"
               + " исключения вне метода: " + e);
     }
}
```

Вызываемый метод **connectFile()** может (при отсутствии файла на диске или при аргументе **null**) генерировать исключение, перехватываемое обработчиком. В результате этого объект непроверяемого исключения **IllegalArgumentException**, как подкласса класса **RuntimeException**, передается обработчику исключений в методе **main()**.

В случае генерации проверяемого исключения компилятор требует обработки исключения в методе или отказа от нее с помощью инструкции **throws**.

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. Для исключений, являющихся подклассами класса **RuntimeException** (unchecked) и используемых для отображения программных ошибок, при выполнении приложения **throws** в объявлении должен отсутствовать.

Ключевое слово finally

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

```
try {/*код, который может вызвать исключение*/}
catch(Exception1 e1) {/*обработка исключения e1*/}//необязателен
catch(Exception2 e2) {/*обработка исключения e2*/}//необязателен
finally {/*выполняется или после try, или после 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 инструкции вида **return**, **break**, **continue**. Приведем пример:

```
/* пример # 4 : выполнение блоков finally: StudentFinally.java */
package chapt08;

public class StudentFinally {
     private static int age;

     public static void setAge(int age) {
          try {
          //реализация
               if(age <= 0)
               throw new RuntimeException("не бывает");
          } finally {
             System.out.print("освобождение ресурсов");
          //реализация
          }
             System.out.print("конец метода");
     }
     public static int getAgeWoman() {
          try {
               return age - 3;
          } finally {
               return age;
          }
     }
     public static void main(String[] args) {
          try {
          setAge(23);
          setAge(-5);
          } catch (RuntimeException e) {
               e.printStackTrace();
          }
```

```
            System.out.print(getAgeWoman());
        }
}
```

В методе **setAge()** из-за генерации исключения происходит преждевременный выход из блока **try**, но до выхода из функции выполняется раздел **finally**. Метод **getAgeWoman()** завершает работу выполнением стоящего в блоке **try** оператора **return**, но и при этом перед выходом из метода выполняется код блока **finally**.

Собственные исключения

Разработчик может создать собственное исключение как подкласс класса **Exception** и затем использовать его при обработке ситуаций, не являющихся исключениями с точки зрения языка, но нарушающих логику вещей. Например, появление объектов типа **Human** (Человек) с отрицательным значением поля **age** (возраст).

```
/* пример # 5 : метод, вызывающий исключение, созданное программистом:
 RunnerLogic.java  */
package chapt08;

public class RunnerLogic {
     public static double salary(int coeff)
                                   throws SalaryException {
          double d;
          try {
               if((d = 10 - 100/coeff) < 0)
               throw new SalaryException("negative salary");
               else return d;
          } catch (ArithmeticException e) {
               throw new SalaryException("div by zero", e);
          }
     }
     public static void main(String[] args) {
          try {
               double res = salary(3); //или 0, или 71;
          } catch (SalaryException e) {
               System.err.println(e.toString());
               System.err.println(e.getHiddenException());
          }
     }
}
```

При невозможности вычислить значение генерируется объект **ArithmeticException**, обработчик которого, в свою очередь, генерирует исключение **SalaryException**, используемое в качестве собственного исключения. Он принимает два аргумента. Один из них – сообщение, которое может быть выведено в поток ошибок; другой – реальное исключение, которое привело к вызову нашего исключения. Этот код показывает, как можно сохранить другую ин-

формацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину вызова **SalaryException**, он всего лишь должен вызвать метод **getHiddenException()**. Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки **SalaryException**.

```
/* пример # 6 : собственное исключение: SalaryException.java */
package chapt08;

public class SalaryException extends Exception {
     private Exception _hidden;

     public SalaryException(String er) {
          super(er);
     }
     public SalaryException(String er, Exception e) {
          super(er);
          _hidden = e;
     }
     public Exception getHiddenException() {
          return _hidden;
     }
}
```

Разработчики программного обеспечения стремятся к высокому уровню повторного использования кода, поэтому они постарались предусмотреть и закодировать все возможные исключительные ситуации. Поэтому при реальном программировании можно вполне обойтись без создания собственных классов исключений.

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 свой блок **throws** все классы исключений или их суперклассы из блока **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. Тогда методы логики пріложения, 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 полиморфный метод компилятор выдает сообщение об ошибке.

```
/* пример # 7 : полиморфизм и исключения: Stone.java: WhiteStone.java:
BlackStone.java: StoneLogic.java */
package chapt08;

class Stone { //старый класс
     public void build() throws FileNotFoundException {
                    /* реализация*/ }
}
class WhiteStone extends Stone {//старый класс
     public void build() {
          System.out.println("белый каменный шар");
     }
}
public class StoneLogic {//старый класс
     public static void infoStone(Stone stone) {
     try {
       stone.build();//обработка IOException не предусмотрена
     } catch(FileNotFoundException e) {
            System.err.print("файл не найден");
      }
    }
}
class BlackStone extends Stone {//новый класс
     public void build() throws IOException{//ошибка компиляции
          System.out.println("черный каменный шар");
          /* реализация*/
     }
}
```

Если же при объявлении метода суперкласса инструкция **throws** присутствует, то в подклассе эта инструкция может вообще отсутствовать или в ней могут быть объявлены любые исключения, являющееся подклассами исключения из блока **throws** метода суперкласса

Второе правило позволяет защитить программиста от возникновения неизвестных ему исключений при создании объекта.

```
/* пример # 8 : конструкторы и исключения: FileInput.java: SocketInput.java */
package chapt08;
import java.io.FileNotFoundException;
import java.io.IOException;

class FileInput {//старый класс
      public FileInput(String filename)
                  throws FileNotFoundException {
            //реализация
      }
}
class SocketInput extends FileInput {
//старый конструктор
      public SocketInput(String name)
```

```
                    throws FileNotFoundException {
            super(name);
            //реализация
        }
/старый конструктор
        public SocketInput() throws IOException {
            super("file.txt");
            //реализация
        }
/новый конструктор
        public SocketInput(String name, int mode) {/*ошибка
                                                    компиляции*/
            super(name);
            //реализация
        }
    }
```

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 **FileInput** не было бы никаких сообщений о возможности генерации исключения, и при возникновении исключительной ситуации ее источник было бы трудно идентифицировать.

Отладочный механизм assertion

Борьба за качество программ ведется всеми возможными способами. На этапе отладки найти неявные ошибки в функционировании приложения бывает довольно сложно. Например, в методе, устанавливающем возраст пользователя, информация о возрасте извлекается из внешних источников (файл, БД), и в результате получается отрицательное значение. Далее неверные данные влияют на результат вычисления среднего возраста пользователей и т.д. Определять и исправлять такие ситуации позволяет механизм проверочных утверждений (assertion). При помощи этого механизма можно сформулировать требования к входным, выходным и промежуточным данным методов классов в виде некоторых логических условий.

Попытка обработать ситуацию появления отрицательного возраста может выглядеть следующим образом:

```
        int age = ob.getAge();
            if (age >= 0) {
            // реализация
            } else {
            // сообщение о неправильных данных
            }
```

Теперь механизм assertion позволяет создать код, который будет генерировать исключение на этапе отладки проверки постусловия или промежуточных данных в виде:

```
        int age = ob.getAge();
            assert (age >= 0): "NEGATIVE AGE!!!";
            // реализация
```

Правописание инструкции **assert**:

```
assert (boolexp): expression;
assert (boolexp);
```

Выражение **boolexp** 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);
// …
   switch (mono) {
         case WHITE : // …
               break;
         case BLACK : // …
               break;
         default :
               assert false : "Colored!";
   }
```

Создателями языка не рекомендуется использовать assertion при проверке параметров **public**-методов. В таких ситуациях лучше генерировать исключения одного из типов: **IllegalArgumentException**, **NullPointerException** 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.

Assertion можно включать для отдельных классов и пакетов при запуске виртуальной машины в виде:

```
java –enableassertions MyClass
```

или

```
java –ea MyClass
```

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

Задания к главе 8

Вариант A

Выполнить задания на основе варианта A главы 4, контролируя состояние потоков ввода/вывода. При возникновении ошибок, связанных с корректностью выполнения математических операций, генерировать и обрабатывать исключительные ситуации. Предусмотреть обработку исключений, возникающих при нехватке памяти, отсутствии требуемой записи (объекта) в файле, недопустимом значении поля и т.д.

Вариант B

Выполнить задания из варианта B главы 4, реализуя собственные обработчики исключений и исключения ввода/вывода.

Тестовые задания к главе 8

Вопрос 8.1.

Дан код:

```
class Quest1{
      int counter;
      java.io.OutputStream out;
   Quest1(){/* инициализация out */}
   float inc() {
      try { counter++;
      out.write(counter); }
//комментарий
}}
```

Какой код достаточно добавить в метод **inc()** вместо комментария, чтобы компиляция прошла без ошибок? (выберите два).

1) `catch (java.io.OutputStreamException e){};`
2) `catch (java.io.IOException e){};`
3) `catch (java.io.OutputException e){};`
4) `finally{};`
5) `return counter;`
6) `return;.`

Вопрос 8.2.

Какое значение будет возвращено при вызове **meth(5)**?

```
public int meth(int x) {
      int y = 010;                  //1
      try { y += x;                 //2
if(x<=5) throw new Exception();     //3
      y++; }                        //4
      catch(Exception e) { y--; }   //5
return y; }                         //6
```

1) 12;
2) 13;
3) 14;
4) 15;
5) ошибка компиляции: невыполнимый код в строке 4.

Вопрос 8.3.

Какое значение будет возвращено при вызове **meth(12)**, если при вызове **mexcept(int x)** возникает исключительная ситуация **ArithmeticException**?

```
int meth(int x) {
      int count=0;
      try { count += x;
            count += mexcept(count);
            count++;
      }catch(Exception e) {
            --count;
      return count;
      }
```

```
      finally {
            count += 3;
      return count;
      }
}
```

1) 11;
2) 12;
3) 13;
4) 14;
5) ошибка компиляции из-за отсутствия **return** после блока **finally.**

Вопрос 8.4.

Какое из следующих определений метода **show()** может законно использоваться вместо комментария //КОД в классе **Quest4**?

```
class Base{
      public void show(int i) {/*реализация*/}
}
public class Quest4 extends Base{
     //КОД
}
```

1) **void** show (**int** i) **throws** Exception {/*реализация*/}
2) **void** show (**long** i) **throws** IOException {/*реализация*/}
3) **void** show (**short** i){ /*реализация*/}
4) **public void** show (**int** i) **throws** IOException {/*реализация*/}

Вопрос 8.5.

Дан код:

```
import java.io.*;
public class Quest5 {
    //ОБЪЯВЛЕНИЕ ioRead()
public static void main(String[] args) {
      try {
            ioRead();
      }catch(IOException e){}
      }}
```

Какое объявление метода **ioRead()** должно быть использовано вместо комментария, чтобы компиляция и выполнение кода прошли успешно?

1) **private static void** ioRead() **throws** IOException{};
2) **public static void** ioRead() **throw** IOException{};
3) **public static void** ioRead(){};
4) **public static void** ioRead() **throws** Exception{}.