9

Исключения (Exceptions) и утверждения (Assertions)

ОБСУЖДАЕМЫЕ ВОПРОСЫ

1. Введение в обработку исключений (Exceptions).
2. Категории исключений.
3. Проверяемые исключения.
4. Обработка исключительных ситуаций.
5. Переопределение методов и исключения.
6. Утверждения (Assertions).
7. Управление выработкой утверждений

ИСКЛЮЧЕНИЯ (EXCEPTIONS)

Исключения позволяют обрабатывать неординарные, с точки зрения выполняющейся программы, ситуации такие, как передача неправильных параметров, сетевые сбои и отсутствие запрашиваемого файла. Такие ситуации могут быть неожиданными для пользователя, но для разработчика они прогнозируемы и требуют специального внимания.
Процедурные языки первых поколений зачастую не предусматривали никаких штатных средств для обработки таких событий – это было исключительной ответственностью разработчика.

В современных языках существуют хорошо проработанные механизмы для обработки исключений. Обработка исключений в приложениях на языке Java является неотъемлемой частью самого приложения.

КАТЕГОРИИ ИСКЛЮЧЕНИЙ

```
Throwable
    |
    |-- Error
    |      |-- VirtualMachineError
    |      |         |-- StackOverflowError
    |      |         |-- OutOfMemoryError
    |      |         `-- . . .
    |      `-- AssertionError
    |
    `-- Exception
           |-- RuntimeException
           |         |-- NullPointerException
           |         |-- ArithmeticException
           |         |--
IllegalArgumentException
           |         `-- . . .
           |-- IOException
           |         |-- FileNotFoundException
           |         `-- . . .
           |-- SQLException
           `-- . . .
```

При возникновении исключительного события создается объект некоторого класса, который содержит информацию о событии.

Это объект передается обработчику исключения или обрабатывается системой, исполняющей программу.

Все классы, порождающие объекты исключений имеют общего предка, класс **Throwable**.

ТИПЫ ИСКЛЮЧЕНИЙ

Серьезные проблемы, которые обычно трактуются, как фатальные ситуации и отражают программные «баги» являются непроверяемыми исключениями. Фатальные ситуации представлены классом `Error`.

Возможные программные «баги», которые в большинстве случаев требуют исправления кода разработчиком, в качестве общего предка имеют класс `RuntimeException`. Эти исключения являются непроверяемыми в том смысле, что компилятор не требует от разработчика их обрабатывать. Этот последний факт приводит к тому, что в специфических случаях некоторые исключения приложения порождаются от `RuntimeException`.

Во время разработки, во избежание конфликтов с компилятором, следует неукоснительно придерживаться правил, касающихся работы с проверяемыми исключениями, сверяясь с документацией на соответствующий API.

ПРОВЕРЯЕМЫЕ ИСКЛЮЧЕНИЯ

Во время разработки, во избежание конфликтов с компилятором, следует неукоснительно придерживаться правил, касающихся работы с проверяемыми исключениями, сверяясь с документацией на соответствующий API.
Синтаксис блока обрабатывающего исключения:

```
try {
       // код, который потенциально может выбрасывать
      //  одно или больше исключений
} catch (SomeException e1) {
      // код, который должен выполняться, если
     // будет выброшено исключение SomeException
} catch (Exception e2) {
       // код выполняющийся, если будет выброшено
      // исключение, являющееся объектом типа Exception
     // или любым, не обработанным выше его потомком
} finally {
  // код, который должен выполняться в любом случае
}
```

ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Конструкция `try-catch` должна иметь только один блок `try` и хотя бы один из блоков: `catch`, или `finally`. То есть блок `try` не может быть сам по себе, ему требуются спутники.
Блоков `catch` может быть множество, однако их надо располагать обратно порядку дерева наследования – сначала располагаем потомков, затем предков. Как вы понимаете, в противном случае до некоторых блоков `catch` дело никогда не дойдет, так как обработчики родительских классов перехватывают все события с потомками.

Блок `finally` может быть только один (если он есть) и всегда стоит последним. Он и срабатывает последним. Он срабатывает в обязательным порядке, даже если вы в блоке `catch` поставите оператор `return`. Он не сработает только тогда, когда вы тем или иным способом организуете выход из JVM, например, вызвав `System.exit(n)`.

ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ (ПРОДОЛЖЕНИЕ)

Если исключение не обработано в текущем блоке `try-catch`, оно выбрасывается в вызывающий метод.

Если исключение приходит в метод `main()` и не обрабатывается и в нем, выполнение программы завершается аварийно.

При принятии решения о том, как поступить в некотором методе в связи рассмотрением возможности потенциального выбрасывания некоторого исключения, следует руководствоваться простыми правилами:

- Если вы можете написать код, который может снять причину исключения так, чтобы программа могла дальше работать, пишите его в соответствующем блоке **catch**.

РЕКОМЕНДАЦИИ ПО ОБРАБОТКЕ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ (ПРОДОЛЖЕНИЕ)

- В некоторых случаях, перехватив одно исключение, можно в блоке `catch` выбросить другое примерно следующим образом.

```
catch( UnsupportedCallbackException e ) {
        . . .
        throw new LoginException( e );
    }
```

- Если вы не желаете обрабатывать проверяемое исключение в данном методе, а желаете переложить ответственность обработки на вызывающий метод, опишите выбрасывание данного исключения в клаузе `throws` данного метода:

```
void trouble() throws IOException, SomeException {
  ...
 }
```

Примечание: Если в клаузе `throws` вы предусмотрели выбрасывание исключения `IOException`, нет смысла в ней описывать каких-либо его потомков, например `FileNotFoundException`, поскольку их выбрасывание обусловлено описанием предка.

РЕКОМЕНДАЦИИ ПО ОБРАБОТКЕ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ (ПРОДОЛЖЕНИЕ)

- Если речь о наследнике `RuntimeException` декларировать его выбрасывание не нужно, так как это непроверяемое исключение. В некоторых случаях вы будете перехватывать и обрабатывать потомков `RuntimeException`. Однако, такое действие трудно порекомендовать для исключений типа `ArrayIndexOutOfBoundsException`. При появлении таких исключений в процессе разработки надо просто внести исправления в код.

- Проверяемые исключения вы обязаны либо обработать, либо явно описать в клаузе `throws`. Если вы не сделаете ни того, ни другого, получите ошибку компиляции.

РЕКОМЕНДАЦИИ ПО ОБРАБОТКЕ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ (ПРОДОЛЖЕНИЕ)

- Никогда не используйте код, который «проглатывает» все исключения «молча»:

```
 . . .
try { … операторы …
}
// перехват практически любых исключений
catch(Exception ex) {} // нет никакой реакции
```

Игнорировать исключения совсем может понадобиться разве что при неаккуратно написанном коде. Если возникла необходимость проигнорировать исключения – правильным будет ограничить список игнорируемых исключений только минимально достаточными и выполнить журналирование, используя метод Throwable.printStackTrace.

ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ И ИСКЛЮЧЕНИЯ

При переопределении родительского метода в рамках полиморфизма, переопределенный метод **может**:

- Не выбрасывать никаких исключений.
- Выбрасывать одно или больше исключений из тех, что описаны в клаузе `throws` переопределяемого метода родительского класса.
- Выбрасывать один или боле субклассов от исключений, выбрасываемых методом родительского класса.

Переопределенный метода **НЕ может**:

- Выбрасывать дополнительные исключения по отношению к тем, что выбрасывает переопределяемый метод.
- Выбрасывать суперклассы исключений по отношению к тем, что выбрасывает переопределяемый метод.

ПРИМЕР ИСПОЛЬЗОВАНИЯ ИСКЛЮЧЕНИЙ ПРИ ПЕРЕОПРЕДЕЛЕНИИ МЕТОДОВ

```
public class ParentA {
   public void methodF() throws IOException {
      // некоторая работа с файлом
   }
 }
 public class TestB extends ParentA {
   public void methodF() throws EOFException {
      // некоторая работа с файлом
   }
 }
 public class TestC extends ParentA {
   public void methodF() throws Exception {     // ОШИБКА !!!
      // некоторая работа с файлом
   }
 }
```

СОЗДАНИЕ ИСКЛЮЧЕНИЙ ПРИЛОЖЕНИЯ

```
public class ServerTimedOutException extends Exception {
    private int port;
    public ServerTimedOutException( String message, int port) {
        super( message );
        this.port = port;
     }
    public int getPort() {
        return port;
     }
 }
```

НЕКОТОРЫЕ МЕТОДЫ КЛАССОВ ИСКЛЮЧЕНИЙ

Для того, чтобы получить текст сообщения можно воспользоваться методом `getMessage()`, унаследованным от класса `Throwable`. Выбросить исключение приложения можно также, как и те, что предусмотренные в пакетах стандартных классов.

```
int port = 80;
 if( ! successful ) {
   throw new ServerTimedOutException( "Could not connect", port );
  }
```

Печать стека вызовов в момент выбрасывания исключения:

```
try {
      // код, который потенциально может выбрасывать исключение
 } catch (SomeException e1) {
     e1.printStackTrace();
 }
```

УТВЕРЖДЕНИЯ (ASSERTIONS)

«Утверждение» (*Assertion)* – это предложение в языке программирования Java, которое позволяет тестировать предположения разработчика об отдельных аспектах разрабатываемой программы.

Например, если некоторая целочисленная переменная из некоторой библиотеки классов в силу логики приложения не должна никогда получать отрицательного значения (и обычно не получает), то разработчик может утверждать (assert) в любом месте программы данная переменная имеет положительное значение. Однако пользователь библиотеки может и не согласиться с данным мнением и тогда приложение, использующее эту библиотеку, просто перестанет правильно работать.

СИНТАКСИС УТВЕРЖДЕНИЯ

Иногда может потребоваться вывод значения для всех ключей **ResourceBundle**.

Метод **getKeys()** возвращает список всех ключей в виде объекта типа **Enumeration**. Далее можно просмотреть в цикле список всех ключей и выбрать для них значения для текущего **ResourceBundle**:

```
assert <булевское_выражение>;
assert <булевское_выражение> : <detail_expression>;
```

Если **<булевское_выражение>** вырабатывает значение **false**, выбрасывается исключение **AssertionError**.
Выражение во втором аргументе **<detail_expression>** конвертируется в строку и используется в качестве описания в сообщении для **AssertionError**.

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ

Пример 1:

```
if (i % 3 == 0) {
        ...
    } else if (i % 3 == 1) {
        ...
    } else {
        assert i % 3 == 2 : ("i = " + i);
        ...
    }
```

Пример 2:

```
void foo() {
        for (...; ...: ...) {
            . . .
            if (...) return;
        }
        assert false; // Выполнение никогда не должно
                      // попадать в это место !
    }
```

УПРАВЛЕНИЕ ВЫРАБОТКОЙ УТВЕРЖДЕНИЙ

По умолчанию проверка утверждений отключена.

При этом код работает так, как будто бы никаких утверждений и нет.

Включение assertions можно осуществить при старте приложения с помощью команд, подобных следующим:

```
java -enableassertions AppClass
```

или

```
java -ea AppClass
```

ИТОГИ

- Рассмотрен синтаксис исключений и механизм их работы.
- Иерархия наследования объектов исключений.
- Выполнен анализ способов использования и и рассмотрены варианты обработки.
- Рассмотрен пример использования исключений при переопределении методов
- Создание исключений приложения
- Рассмотрен синтаксис утверждений (Assertions) и примеры их использования
- Управление выработкой утверждений