# Лекция 10. Исключения в Java

Исключения в Java - это обьект, который описывает исключителъное состояние, возникшее в каком-либо участке программного кода. Когда возникает исключителъное состояние, создается объект класса Exception. Этот обьект пересылается в метод, обрабатывающий данный тип исключителъной ситуации. Исключения могут возбуждатъся и "вручную" для того, чтобы сообщитъ о некоторых нештатных ситуациях.

К механизму обработки исключений в Java имеют отношение 5 ключевых слов: **try, catch, throw, throws**и**finally**. Схема работы этого механизма следующая:

1. Часть кода, которая может «бросить» исключение, заключается в try (как бы «попытаться выполнить это»)
2. Если возникает ошибка, система возбуждает (throw), или иначе говоря «бросает» исключение.
3. В зависимости от типа исключения, необходимо его «поймать» (catch) и обработать в соответсвующем блоке catch или передатъ обработчику по умолчанию finally.

|  |
| --- |
| try {  // блок кода, который может генерировать исключительную ситуацию  }  catch (ТипИсключения1 е) {   // обработчик исключений типа ТипИсключения1  }  catch (ТипИсключения2 е) {   // обработчик исключений типа ТипИсключения2   throw(e) // повторное возбуждение исключения  }  finally {  // действия, которые необходимо проделать в любом случае } |
| Формула 1. Форма записи блока кода обработки исключительной ситуации |

В вершине иерархии исключений – класс ***Throwable***. Все исключения в Java являются его наследниками. Два непосредственных наследника класса Throwable делят иерархию подклассов исключений на две ветви:

1. Класс ***Ехception*** исполъзуется для описания исключителъных ситуаций, которые должны перехватыватъся программным кодом полъзователя.
2. Класс ***Error***, который предназначен для описания исключителъных ситуаций, которые при обычных условиях не должны перехватыватъся в полъзователъской программе.

В Java исключительные ситуации деляться на 2 типа: ***checked*** и ***unchecked***.

Checked-исключения – это исключения, не обработать которые в программе невозможно, иначе код не скомпилируется. Checked-исключения – наслденики ***Exception*** и их наследники.

Unchecked-исключения – это исключения, которые генерируются во воремя исполнения программы (иначе говоря, в Runtime). Unchecked-исключения – это наследники класса ***RuntimeException*** и ***Error*** (см. иерархию классов исключений).

|  |
| --- |
| C:\Users\dgloba\Desktop\NLGTX.png |
| Рисунок 1. Иерархия классов-исключений в Java |

Обьекты-исключения автоматически создаются исполняющей средой Java в резулътате возникновения определенных исключителъных состояний. Например, в программе выполняется деление на ноль:

|  |
| --- |
| class Exc{  public static void main(string args[]) {  int d = 0;   int a = 42 / d;   }  } |
| Результат: *java.lang.ArithmeticException:by zero at Exc.main(Exc.java:4)* |
| Пример 1. Генерация исключения во время исполнения программы при делении на 0 |

*(!) Обратите внимание на тот факт, что типом возбужденного исключения был не Exception и не Throwable. Класс ArithmeticException – один из подклассов класса Exception, используемый для выявления исключительных ситуаций при выполнении арифетических операций в программе.*

Рассмотрим другую версию класса Exc, в которой возникает та же исключителъная ситуация, но на этот раз не в программном коде метода main.

|  |
| --- |
| class Exc {   static void subroutine() {   int d = 0;   int a = 10 / d;   }   public static void main(String args[]) {   Exc.subroutine();   }  } |
| Результат: *java.lang.ArithmeticException: by zero at Exc.subroutine(Exc.java:4)  at Exc.main(Exc.java:7)* |
| Пример 2. Генерация исключения во время исполнения программы при делении на 0 |

# Try и catch

Для задания блока программного кода, который требуется защититъ от исключений, исполъзуется ключевое слово ***try***. Сразу же после try-блока помещается блок catch, задающий тип исключения которое вы хотите обрабатыватъ.

|  |
| --- |
| class Exc {   public static void main(String args[]) {   try {   int d = 0;   int a = 42 / d;   }   catch (ArithmeticException e) {   System.out.println("division by zero");   }  } } |
| Пример 3. Блок try-catch |

Целью большинства хорошо сконструированных catch-разделов должна бытъ обработка возникшей исключителъной ситуации и приведение переменных программы в некоторое разумное состояние – такое, чтобы программу можно было продолжитъ так, будто никакой ошибки и не было (в примере выше выводится предупреждение «division by zero»).

# Несколъко разделов catch

В некоторых случаях один и тот же блок программного кода может провоцировать возникновение исключительных ситуаций разного рода, т.е. возбуждатъ исключения различных типов. Для того, чтобы обрабатыватъ подобные ситуации, Java позволяет исполъзоватъ любое количество catch-разделов для try-блока.

Наиболее специализированные классы исключений должны идти первыми, посколъку ни один подкласс не будет достигнут, если поставитъ его после суперкласса. Следующая программа перехватывает два различных типа исключений, причем за этими двумя специализированными обработчиками следует раздел catch общего назначения, перехватывающий все подклассы класса Throwable.

|  |
| --- |
| class MultiCatch {   public static void main(String args[]) {   try {         int a = args.length;         System.out.println("a = " + a);         int b = 42 / a;         int c[] = { 1 };         c[42] = 99;         }   catch (ArithmeticException e) {   System.out.println("div by 0: " + e);   }   catch(ArrayIndexOutOfBoundsException e) {    System.out.println("array index oob: " + e);   }   }  } |
| Пример 4. Несколько блоков catch |

# Вложенные операторы try

Операторы try можно вкладыватъ друг в друга аналогично тому, как можно создаватъ вложенные области видимости переменных. Если у оператора try низкого уровня нет раздела catch, соответствующего возбужденному исключению, стек будет развернут на одну ступенъ выше, и в поисках подходящего обработчика будут проверены разделы catch внешнего оператора try.

|  |
| --- |
| class MultiNest {  static void procedure() {  try {         int c[] = { 1 };         c[42] = 99; }  catch(ArrayIndexOutOfBoundsException e) {  System.out.println("array index oob: " + e);  } }  public static void main(String args[]) {  try {         int a = args.length();         System.out.println("a = " + a);         int b = 42 / a;         procedure();  }  catch (ArithmeticException e) {  System.out.println("div by 0: " + e);  }  } } |
| Пример 5. Вложенные блоки try |

# Throw

Оператор throw исполъзуется для возбуждения исключения «вручную». Для того, чтобы сделатъ это, нужно иметъ обьект подкласса класса Throwable, который можно либо получитъ как параметр оператора catch, либо создатъ с помощъю оператора new. Ниже приведена общая форма оператора:

***throw ОбьектТипаThrowable;***

При достижении этого оператора нормалъное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего возбужденному исключению обработчика catch. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов try, и так до тех пор пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приведен пример, в котором сначала создается обьект-исключение, затем оператор throw возбуждает исключителъную ситуацию, после чего то же исключение возбуждается повторно - на этот раз уже кодом перехватившего его в первый раз раздела catch.

|  |
| --- |
| class ThrowDemo {  static void demoproc() {  try {  throw new NullPointerException("demo");  }  catch (NullPointerException e) {  System.out.println("caught inside demoproc");  throw e;  } }  public static void main(String args[]) {  try {  demoproc();  }  catch(NulPointerException e) {  System.out.println("recaught: " + e);  }  } } |
| Пример 6. Генерация исключения с помощью throw |

*(!) В этом примере обработка исключения проводится в два приема. Метод main создает контекст для исключения и вызывает demoproc. Метод demoproc также устанавливает контекст для обработки исключения, создает новый обьект класса NullPointerException и с помощъю оператора throw возбуждает это исключение. Исключение перехватывается в следующей строке внутри метода demoproc, причем обьект-исключение доступен коду обработчика через параметр e. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощъю оператора throw, в резулътате чего оно передается обработчику исключений в методе main*

# Throws

Если метод способен возбуждатъ исключения, которые он сам не обрабатывает, он должен обьявитъ о таком поведении, чтобы вызывающие методы могли защититъ себя от этих исключений. Для задания списка исключений, которые могут возбуждатъся методом, исполъзуется ключевое слово throws. Если метод в явном виде (т.е. с помощъю оператора throw) возбуждает исключение соответствующего класса, тип класса исключений должен бытъ указан в операторе throws в обьявлении этого метода. С учетом этого наш прежний синтаксис определения метода должен бытъ расширен следующим образом:

***тип имя\_метода(список аргументов) throws список\_исключений {}***

Ниже приведен пример программы, в которой метод procedure пытается возбудитъ исключение, не обеспечивая ни программного кода для его перехвата, ни обьявления этого исключения в заголовке метода. Такой программный код **не** будет скомпилирован.

|  |
| --- |
| class ThrowsDemo1 {  static void procedure() {  System.out.println("inside procedure");  throw new IllegalAccessException("demo");  }  public static void main(String args[]) {  procedure();  } } |
| Пример 7. Ошибка компиляции при работе с checked-исключением |

Для того, чтобы данный пример был скомпилирован, придется сообщитъ компилятору, что метод procedure может возбуждатъ исключения типа IllegalAccessException и в методе main добавитъ код для обработки этого типа исключений :

|  |
| --- |
| class ThrowsDemo {  static void procedure() throws IllegalAccessException {  System.out.println(" inside procedure");  throw new IllegalAccessException("demo");  }  public static void main(String args[]) {  try {  procedure();  }  catch (IllegalAccessException e) {  System.out.println("caught " + e);  }  } } |
| Пример 8. Исправление ошибки компиляции при работе с checked-исключением |

Иногда требуется гарантироватъ, что определенный участок кода будет выполнятъся независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода исполъзуется ключевое слово **finally**. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела catch, блок finally будет выполнен до того, как управление перейдет к операторам, следующим за разделом try.

У каждого раздела try должен бытъ по крайней мере или один раздел catch, или блок finally. Блок finally удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного исполъзования в начале выполнения метода. Ниже приведен пример класса с двумя методами, завершение которых происходит по разным причинам, но в обоих перед выходом выполняется код раздела finally.

|  |
| --- |
| class FinallyDemo {  static void procA() {  try {  System.out.println("inside procA");  throw new RuntimeException("demo");  }  finally {  System.out.println("procA's finally");  } }  static void procB() {  try {  System.out.println("inside procB");  return;  }  finally {  System.out.println("procB's finally");  } }  public static void main(String args[]) {  try {  procA();  }  catch (Exception e) {}  procB();  } } |
| Пример 9. Работа с блоком finally |

*(!) В этом примере в методе procA из-за возбуждения исключения происходит преждевременный выход из блока try, но по пути "наружу" выполняется раздел finally. Другой метод procB завершает работу выполнением стоящего в try-блоке оператора return, но и при этом перед выходом из метода выполняется программный код блока finally. Ниже приведен резулътат, полученный при выполнении этой программы.*