EPAM Systems, RD Dep.

Конспект и раздаточный материал

JAVA.SE.05 Exceptions and Errors

REVISION HISTORY					
Ver.	**Description of Change**	**Author**	**Date**	**Approved**	
				Name	**Effective Date**
<1.0>	Первая версия	Игорь Блинов	<17.08.2011>		
<2.0>	Вторая версия. Конспект переделан под обновленное содержание материала модуля.	Ольга Смолякова	<09.04.2014>		

СОДЕРЖАНИЕ
JAVA.SE.05 EXCEPTIONS AND ERRORS

Понятие исключения

Исключение — это аварийное состояние, которое возникает в кодовой последовательности во время выполнения.

Другими словами, **исключение** — это ошибка времени выполнения.

В машинных языках, *не поддерживающих обработку исключений*, **ошибки** должны быть **проверены и обработаны вручную** — обычно с помощью кодов ошибки, и т. д.

Обработка исключений в Java переносит управление обработкой ошибок времени выполнения в объектно-ориентированное русло.

Основные принципы обработки исключений

Исключение в языке Java — это **ОБЪЕКТ**, который описывает исключительную (т. е. ошибочную) ситуацию, произошедшую в некоторой части кода.

Когда исключительная ситуация возникает, создается объект, представляющий это исключение, и «вбрасывается» в метод, вызвавший ошибку.
В свою очередь, метод может выбрать, *обрабатывать* ли исключение самому или *передать* его куда-то еще.

В любом случае, в некоторой точке исключение «захватывается» и обрабатывается.

Исключения могут **генерироваться исполнительной системой** Java, или ваш код может сгенерировать их **"вручную"**.

Выбрасываемые исключения касаются *фундаментальных ошибок*, которые нарушают ограничения среды выполнения или правила языка Java.

Исключения, **сгенерированные вручную**, обычно используются, чтобы сообщить вызывающей программе о некоторой *аварийной ситуации*.

Обработка исключений в Java управляется с помощью пяти ключевых слов:

- ***try***,
- ***catch***,
- ***throw***,
- ***throws,***
- ***finally***.

Программные операторы, которые нужно контролировать относительно исключений, содержатся в блоке ***try***.

Если в блоке ***try*** происходит исключение, говорят, что оно выброшено (***thrown***) этим блоком.

Ваш код может перехватить (***catch***) это исключение (используя оператор ***catch***) и обработать его некоторым рациональным способом.

Исключения, генерируемые исполнительной (***run-time***) системой Java, **выбрасываются автоматически**.

Для **"ручного"** выброса исключения используется ключевое слово ***throw***. Любое исключение, которое выброшено из метода, следует определять с помощью ключевого слова ***throws***, размещаемого в заголовочном предложении определения метода.

Любой код, который обязательно должен быть выполнен перед возвратом из ***try***-блока, размещается в ***finally***-блоке, указанном в конце блочной конструкции ***try{... }-catch{... }-finally{...}***.

Общая форма блока обработки исключений:

```
try{
        // блок кода для контроля над ошибками
} catch(ExceptionType1 exOb) {
        // обработчик исключений для ExceptionTypel
} catch (ExceptionType2 exOb) {
        // обработчик исключений для ExceptionType2
}
[finally{
        // блок кода для обработки перед возвратом из try блока
}]
```

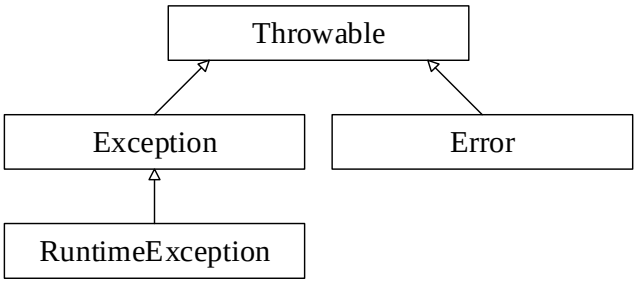
Здесь ExceptionType — тип исключения, которое возникло; exOb — объект этого исключения, finally-блок — не обязателен.

Типы исключений

Все типы исключений являются подклассами встроенного класса **Throwable**.

Throwable представляет собой вершину иерархии классов исключений.

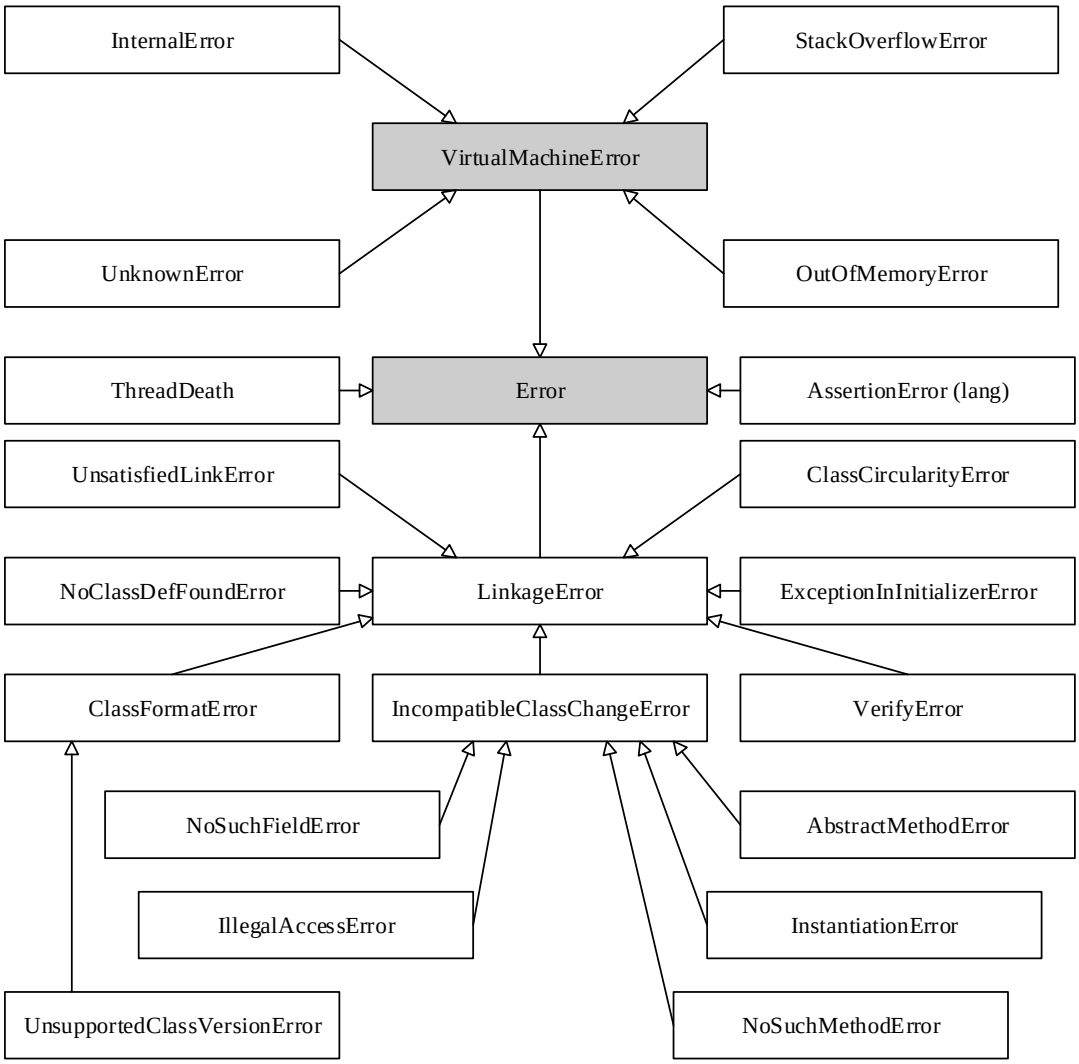
Непосредственно ниже **Throwable** находятся два подкласса, которые разделяют исключения на две различные ветви.



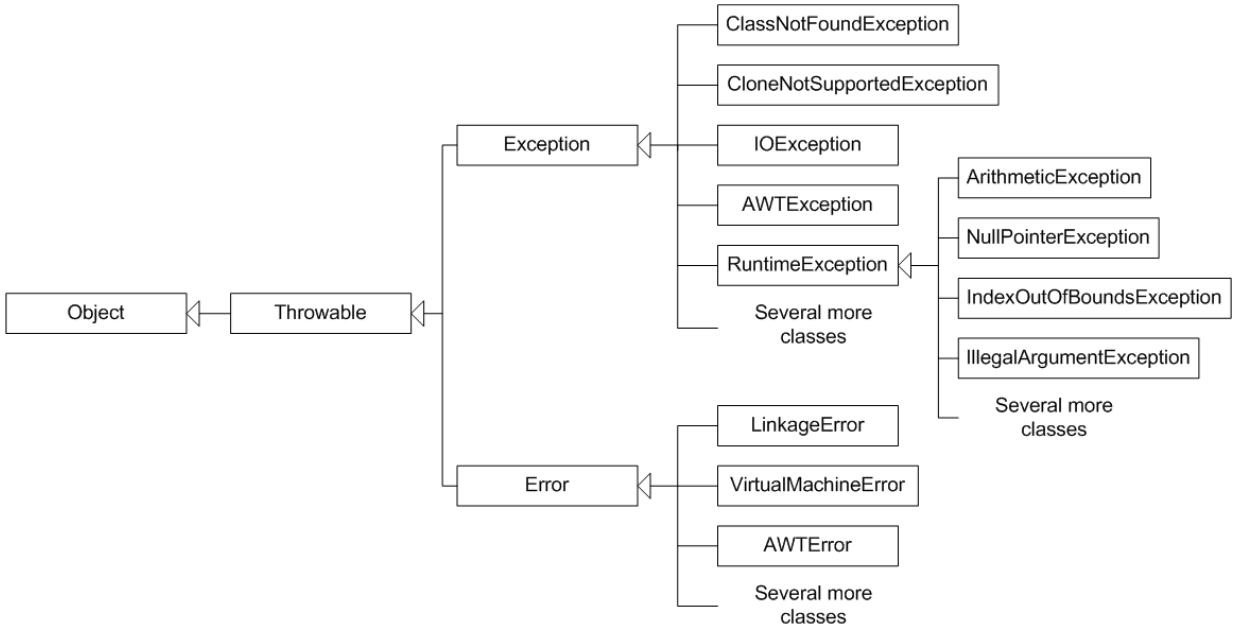
Одна ветвь возглавляется классом **Exception**. Этот класс используется для исключительных состояний, которые должны перехватывать программы пользователя. Это также класс, в подклассах которого вы будете создавать ваши собственные заказные типы исключений.

Другую ветвь возглавляет класс **Error**, определяющий исключения, перехват которых вашей программой при нормальных обстоятельствах не ожидается. Исключения типа **Error** применяются исполнительной системой Java для указания ошибок, имеющих отношение непосредственно к среде времени выполнения.

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.



Исключительные ситуации типа **Exception** - это проверяемые (**checked**) исключения, исключительные ситуации типа **RuntimeException** - непроверяемые (**unchecked**) исключения.



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.

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.

Подклассы непроверяемых исключений Java

Исключение	Значение
ArithmeticException	Арифметическая ошибка типа деления на нуль
ArrayIndexOutOfBoundsException	Индекс массива находится вне границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора, типа ожидания на разблокированном потоке
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока

IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
SecurityException	Попытка нарушить защиту
StringIndexOutOfBoundsException	Попытка индексировать вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

Подклассы проверяемых исключений (*java.lang*)

Исключение	Значение
ClassNotFoundException	Класс не найден
CloneNotSupportedException	Попытка клонировать объект, который не реализует интерфейс Cloneable
IllegalAccessException	Доступ к классу отклонен
InstantiationException	Попытка создавать объект абстрактного класса или интерфейса
InterruptedException	Один поток был прерван другим потоком
NoSuchFieldException	Требуемое поле не существует
NoSuchMethodException	Требуемый метод не существует

Использование операторов try и catch

Неотловленные исключения.

```
package _java._se._02._trycatch;
public class Division {
	public static void main(String[] args) {
		int d = 0;
		int a = 42 / d;
	}
}
```

Когда исполнительная система Java обнаруживает попытку деления на ноль, она создает новый объект исключения и затем *выбрасывает* его.

Перехват исключения.

```
package _java._se._02._trycatch;
```

```
public class DivisionWithTry {
	public static void main(String[] args) {
		int d, a;
		try {
			d = 0;
			a = 42 / d;
			System.out.println("Этот текст никогда не будет напечатан.");
		} catch (ArithmeticException e) {
			System.out.println("Деление на ноль.");
		}
		System.out.println("Уже после блока try-catch.");
	}
}
```

Множественные операторы catch

В некоторых случаях на одном участке кода может возникнуть более одного исключения. После того как этот **catch**-оператор выполнится, другие — обходятся, и выполнение продолжается после блока **try/catch**.

```
package _java._se._02._trycatch;
public class MultiCatch {
	public static void main(String[] args) {
		int a;
		try {
			a = args.length;
			int b = 42 / a;
			int[] c = new int[a];
			c[a] = 666;
		} catch (ArithmeticException e) {
			System.out.println("Деление на ноль." + e);
		} catch (ArrayIndexOutOfBoundsException e) {
			e.printStackTrace();
		}
		System.out.println("Уже после блока try-catch-catch.");
	}
}
```

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

```
/* суперкласс Exception перехватит объекты всех своих подклассов */
catch (Exception e) {
}
/* не может быть вызван, поэтому возникает ошибка компиляции */
catch (ArithmeticException e) {
}
```

Мультиобработчик.

В *Java 7* стало возможным использовать один обработчик для перехвата сразу нескольких исключений (*multi-catch*).

Для одновременной обработки сразу нескольких исключений типы исключений в блоке *catch* разделяются оператором *ИЛИ* (*OR*).

```
try{
		// ...
```

```
}
catch(IOException | ArrayIndexOutOfBoundsException e){
    System.out.println("Error: " + e.getMessage());
}
```

Вложенные операторы try

Операторы **try** могут быть вложенными.

Один **try**-оператор может находиться внутри блока другого оператора **try**.

При входе в блок **try** контекст соответствующего исключения помещается в стек.

Если внутренний оператор **try** не имеет **catch**-обработчика для специфического исключения, стек раскручивается, и просматривается следующий **catch**-обработчик **try**-оператора.

Процесс продолжается до тех пор, пока не будет достигнут подходящий **catch**-оператор, или пока все вложенные операторы try не будут исчерпаны.

Если согласующегося оператора **catch** нет, то исключение обработает исполнительная система Java.

```
package _java._se._02._trycatch;
public class NestTry {
	public static void main(String args[]) {
		try {
			int a = args.length;
			int b = 42 / a;
			System.out.println("a = " + a);
			function(a);
		} catch (ArithmeticException e) {
			System.out.println("Деление на нуль: " + e);
		}
	}
	public static void function(int a) {
		try {
			if (a == 1)
				a = a / (a - a);
			if (a == 2) {
				int c[] = { 1 };
				c[42] = 99;
			}
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Индекс выходит за границу массива: " + e);
		}
	}
}
```

Автоматическое управление ресурсами

В *Java 7* реализована возможность автоматического закрытия ресурсов в блоке ***try*** с ресурсами (*try-with-resources*).

В операторе *try* открывается ресурс (файловый поток ввода), который затем читается.

При завершении блока *try* данный ресурс автоматически закрывается, поэтому нет никакой необходимости явно вызывать метод *close()* у потока ввода, как это было в предыдущих версиях *Java*.

Автоматическое управление ресурсами возможно только для тех ресурсов, которые реализуют интерфейс *java.lang.AutoCloseable*.

```
package _java._se._02._trycatch;
```

```
import java.io.FileInputStream;
import java.io.IOException;
public class TryWithResources {

	public static void main(String[] args) {
		String filePath = args[0];
		try (FileInputStream in = new FileInputStream(filePath)) {
			int data = 0;
			while ((data = in.read()) != -1) {
				System.out.print("Data: " + data);
			}
		} catch (IOException e) {
			System.out.println("Error: " + e.getMessage());
		}
	}
}
```

Оператор throw и ключевое слово throws

Программа может сама явно выбрасывать исключения, используя оператор **throw**. Общая форма оператора **throw** такова:

```
throw  ThrowableInstance;
```

Здесь ThrowableInstance должен быть объектом типа **Throwable** или подкласса **Throwable**. Простые типы, такие как **int** или **char**, а также не-**Throwable**-классы (типа **String** и **Object**) не могут использоваться как исключения.

Имеется два способа получения **Throwable**-объекта: использование параметра в предложении **catch** или создание объекта с помощью операции **new**.

Если метод способен к порождению исключения, которое он не обрабатывает, он должен определить свое поведение так, чтобы вызывающие методы могли сами предохранять себя от данного исключения.

Это *обеспечивается включением предложения* **throws** в заголовок объявления метода.

Предложение **throws** *перечисляет типы исключений, которые метод может выбрасывать*. Это необходимо для всех исключений, кроме исключений типа **Error, RuntimeException** или любых их подклассов.

Все другие исключения (кроме исключения типа **Error, RuntimeException**), которые метод может выбрасывать, должны быть объявлены в предложении **throws.**

Если данное условие не соблюдено, то произойдет ошибка времени компиляции.

Общая форма объявления метода, которое включает предложение **throws**:

```
type method-name(parameter-list)
throws  exception-list {
		// тело метода
}
```

Здесь *exception-list* — список разделенных запятыми исключений, которые метод может выбрасывать

```
package _java._se._02._trycatch;
public class ThrowsGenerate {
	static void throwOne() throws IllegalAccessException {
		System.out.println("Внутри throwOne.");
		throw new IllegalAccessException("demo");
	}
	public static void main(String args[]) {
		try {
```

```
                throwOne();
            } catch (IllegalAccessException e) {
                System.out.println("Выброс " + e);
            }
        }
}
```

Перехваченное исключение может быть сгенерировано снова.

```
package _java._se._02._trycatch;
public class ThrowTwice {
    public static void main(String args[]) {
        try {
            int a = args.length;
            try {
                if (a == 1) {
                    a = a / (a - a);
                }
                if (a == 2) {
                    int c[] = { 1 };
                    c[42] = 99;
                }
            } catch (ArrayIndexOutOfBoundsException e) {
               System.out.println("Индекс выходит за границу массива: " + e);
               throw e;
               // or // throw new // ArithmeticException("Исключение в
               // catch.");
            }
            int b = 42 / a;
            System.out.println("a = " + a);
        } catch (ArithmeticException e) {
            System.out.println("Деление на ноль: " + e);
        } catch (Exception e) {
            System.out.println("Общий обработчик.");
        }
    }
}
```

Java 7 позволяет передавать "вверх" по стеку вызова исключение более точного типа, если данные типы указаны при объявлении метода.

```
public static void remoteMethod() throws RemoteException{
    try{
        throw new RemoteException("this is RemoteException");
    }
    catch(Exception e){
        throw e;
    }
}
```

Блок finally

Когда исключение выбрасывается, выполнение метода имеет довольно неровный, нелинейный путь, который изменяет нормальное прохождение потока через метод.

В зависимости от того, как кодирован метод, исключение может вызвать даже преждевременный выход из него.

Например, если метод открывает файл для ввода и закрывает его для вывода, то вы вряд ли захотите, чтобы закрывающий файл код был обойден механизмом обработки исключений.

Для реализации этой возможности и предназначено ключевое слово **finally**.

```
package _java._se._02._trycatch;
```

```
public class FinallyUse {
	static void procA() {
		try {
			System.out.println("Внутри procA");
			throw new RuntimeException("demo");
		} finally {
			System.out.println("finally для procA ");
		}
	}
	// Возврат изнутри try-блока
	static void procB() {
		try {
			System.out.println("Внутри procB");
			return;
		} finally {
			System.out.println("finally для procB ");
		}
	}
	// Нормальное выполнение try-блока
	static void procC() {
		try {
			System.out.println("Внутри procC");
		} finally {
			System.out.println("finally procC");
		}
	}
	public static void main(String args[]) {
		try {
			procA();
		} catch (Exception e) {
			System.out.println("Исключение выброшено");
		}
		procB();
		procC();
	}
}
```

Блок finally выполняется в любом случае.

```
package _java._se._02._trycatch;
public class FinallyAndReturn {
	private static int age = 20;

	public static int getAgeWoman() {
		try {
			return age - 3;
		} finally {
			return age;
		}
	}

	public static void main(String[] args) {
		System.out.println(getAgeWoman());
	}
}
```

Создание собственных исключений

Чтобы создать собственное исключение, его класс надо унаследовать от **Throwable** или от его подкласса (чаще всего от класса **Exception**).

Класс **Exception** не определяет никаких собственных методов, а наследует эти методы от класса **Throwable**. Таким образом, всем исключениям, даже тем, что вы создаете сами, доступны методы **Throwable**.

```
package _java._se._02._trycatch;
public class MyException extends Exception {
	private int detail;

	MyException(int a) {
		detail = a;
	}

	public String toString() {
		return "MyException[" + detail + "]";
	}
}
```

Методы, определенные в **Throwable**.

Метод	Описание
Throwable filllnStackTrace ()	Возвращает Throwable-объект, который содержит полную трассу стека. Этот объект может быть выброшен повторно
String getLocalizedMessage()	Возвращает локализованное описание исключения
String getMessage()	Возвращает описание исключения
void printStackTrace()	Отображает трассу стека
void printStackTrace (PrintStream stream)	Посылает трассу стека указанному потоку
void printStackTrace (PrintWriter stream)	Посылает проекцию прямой стека указанному потоку
String toString()	Возвращает string-объект, содержащий описание исключения. Этот метод вызывается из println() при выводе Throwable-объекта

```
package _java._se._02._trycatch;
public class UseHiddenException {
	public static double salary(int coeff) throws HiddenException {
		double d;
		try {
			if ((d = 10 - 100 / coeff) < 0)
				throw new HiddenException("negative salary");
			else
				return d;
		} catch (ArithmeticException e) {
			throw new HiddenException("div by zero", e);
		}
	}

	public static void main(String[] args) {
		try {
```

```
                double res = salary(3); // или 0, или 71;
            } catch (HiddenException e) {
                System.err.println(e.toString());
                System.err.println(e.getHiddenException());
            }
        }
}

package _java._se._02._trycatch;
public class HiddenException extends Exception {
        private Exception _hidden;

        public HiddenException(String er) {
            super(er);
        }

        public HiddenException(String er, Exception e) {
            super(er);
            _hidden = e;
        }

        public Exception getHiddenException() {
            return _hidden;
        }
}
```

Исключения при наследовании

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.

```
package _java._se._02._trycatch;
import java.io.IOException;
public class BaseCl {
          public BaseCl() throws IOException, ArithmeticException {
          }
          public static void methodA() throws IOException {
          }

}

package _java._se._02._trycatch;
import java.io.EOFException;
import java.io.IOException;
public class DerivativeCl extends BaseCl {
        public DerivativeCl() throws EOFException, IOException, ArithmeticException {
            super();
        }

        public static void methodA() throws EOFException {
        }
}
```

```
package _java._se._02._trycatch;
public class DerivativeCl2 extends BaseCl {
	// ошибок компиляции нет
	public DerivativeCl2() throws Exception {
		super();
	}

	// compile error
	public static void methodA() throws Exception {
	}
}
```

Исключения в конструкторе

Если в конструкторе будет выброшено исключение – объект создан не будет.

```
package _java._se._02._trycatch;
public class ConstructorException {
	private int i;

	public ConstructorException(int _i) {
		i = 20 / i;
	}

	public int getI() {
		return i;
	}
}
```

```
package _java._se._02._trycatch;
public class ExceptionInConstructorTest {
	public static void main(String[] args) {
		ConstructorException p = null;
		try {
			p = new ConstructorException(0);
		} catch (ArithmeticException e) {
			System.out.println("Гасим исключение конструктора.");
		}
		System.out.println(p.getI());
	}
}
```

Применение исключений

Обработка исключений обеспечивает мощный механизм управления комплексными программами, обладающими множеством динамических характеристик времени выполнения.

Важно представлять механизм **try-throw-catch**, как достаточно ясный способ обработки ошибок и необычных граничных условий в логике программы.

Когда произойдет *отказ метода*, пусть он *сам выбросит исключение* - это более ясный способ обработки режимов отказа.

Операции обработки исключений Java не нужно рассматривать как общий механизм для нелокального ветвления. Если вы так сделаете, это только запутает ваш код и затруднит его поддержку.