

# Наследование

© Составление, Гаврилов А.В., Будаев Д.С., Стефанов М.А. 2018

Лекция 2.1

Самара  
2020

# План лекции

---

- Наследование классов и создание объектов дочерних классов
- Переопределение методов
- Соккрытие полей
- Завершенные и абстрактные методы и классы
- Описание и применение интерфейсов

# Наследование в Java

---

## Виды наследования

- Класс
  - Расширяет класс *и/или*
  - Реализует интерфейс(ы)
- Интерфейс
  - Расширяет интерфейс(ы)

# Расширение классов

- Класс может расширить **только один класс**
- Расширяющий класс называется **производным** (дочерним, подклассом)
- Расширяемый класс называется **базовым** (родительским, суперклассом)

```
class MyClass1 {  
}  
  
class MyClass2 extends MyClass1 {  
}
```

# Конструкторы дочерних классов

- Вызываются при создании объектов **дочерних** классов
- Могут вызывать **друг друга** по ключевому слову **this ()**
- Могут вызывать конструкторы **базового** класса по ключевому слову **super ()**
- Вызов конструктора суперкласса должен быть первой инструкцией конструктора дочернего класса
- Ключевое слово **super ()** может не использоваться, **только если** в родительском классе **существует конструктор по умолчанию**

# Порядок создания объекта

- Порядок вызова конструкторов:
  - Вызов конструктора **базового** класса
  - Присваивание исходных значений полям объекта посредством выполнения соответствующих **выражений** и **блоков инициализации**
  - Выполнение инструкций **в теле** конструктора (конструкторов)
- Состояние объекта инициализируется «послойно» от **Object** до конкретного класса

# Пример наследования

```
class SuperShow {
    public String str = "SuperStr";

    public void show() {
        System.out.println("Super.show: " + str);
    }
}

class ExtendShow extends SuperShow {
    public String str = "ExtendStr";

    public void show() {
        System.out.println("Extend.show: " + str);
    }
}
```

# И его результат

```
public static void main(String[] args) {  
    ExtendShow ext = new ExtendShow();  
    SuperShow sup = ext;  
    ext.show();  
    sup.show();  
    System.out.println("ext.str = " + ext.str);  
    System.out.println("sup.str = " + sup.str);  
}
```

```
Extend.show: ExtendStr  
Extend.show: ExtendStr  
ext.str = ExtendStr  
sup.str = SuperStr
```

# Совпадение имен методов в родительском и дочернем классах

- Сигнатуры различны

**Перегрузка** – добавляется метод с другими параметрами

- Сигнатуры совпадают

**Переопределение** – замещение версии метода, объявленной в базовом классе, новой, с точно такой же сигнатурой

# Переопределение методов

- При обращении **извне** объекта производного класса к его методу **всегда** вызывается **новая** версия метода
- Доступ к методу базового класса **изнутри** объекта дочернего класса может быть получен с помощью ключевого слова **super**
- **Уровень доступа** метода при переопределении не может сужаться
- Методы **private** не переопределяются

# Переопределение методов

- В предложении **throws** дочернего метода не может быть типов исключений, несовместимых с типами в предложении **throws** родительского метода
- Переопределенный метод может быть снабжен модификатором **abstract**
- Признаки **synchronized**, **native** и **strictfp** могут изменяться произвольно

# Соккрытие полей

- Поля не переопределяются, но скрываются
- Тип поля при соккрытии можно изменить
- Поле базового класса при соккрытии продолжает существовать, но недоступно непосредственно по имени
- Доступ можно получить с помощью ключевого слова `super` либо через `ссылочную переменную родительского типа`
- Имеет право на существование следующая конструкция:  
`(VeryBaseClass) this`

# Служебное слово `super`

- Действует как ссылка на текущий экземпляр по контракту базового класса
- Может быть использовано в теле любого нестатического члена класса
- Формы использования
  - `super(...)`
  - `super.method(...)`
  - `super.field`

# Соккрытие статических членов

- Статические члены не могут быть переопределены, они скрываются
- Обычно для доступа используется имя класса, поэтому проблем не возникает
- Если используется ссылка, то учитывается объявленный тип ссылки, а не фактический тип объекта

# Замечание

---

Важно понимать, что:

- Переопределение методов – фундаментальный механизм ООП, в частности, обеспечивающий полиморфизм
- Соккрытие полей – следствие отсутствия ограничений на имена полей

# Завершенные методы и классы

- Завершенный метод не допускает переопределения

```
final public int getValue();
```

- Завершенный класс не допускает расширения

```
final class MyClass {  
    ...  
}
```

# Абстрактные классы и методы

- Абстрактные методы описывают сигнатуру без реализации

```
abstract public int getValue();
```

- Класс с абстрактными методами **обязан** быть абстрактным

```
abstract class MyClass {...}
```

- Расширяющий класс может перекрыть своими абстрактными родительские реализованные методы
- Абстрактный класс не обязан иметь абстрактные методы
- **Создавать объекты абстрактных типов нельзя!**

# Контракт класса

- Набор методов и полей класса, открытых для доступа извне тем или иным способом, в совокупности с описанием их назначения
- Способ выражения обещаний автора относительно того, на что способен и для чего предназначен созданный им продукт

# Наследование

## Практическое воплощение наследования

- Наследование **контракта** или **типа**, в результате чего производный класс получает тип базового, поэтому может быть использован полиморфным образом
- Наследование способов **реализации**, в результате производный класс приобретает функциональные характеристики базового в виде набора доступных полей и методов

# Понятие интерфейса

- Позволяет описать тип в **полностью абстрактной форме**
- Экземпляры интерфейсов создавать нельзя
- Классы способны реализовывать один или несколько интерфейсов
- Реализация классом интерфейса означает согласие класса на внешний контракт, описываемый реализуемым интерфейсом

# Наследование в Java

## Виды наследования

### ■ Класс

Наследование как внешнего контракта, так и реализации

- Расширяет класс
- Реализует интерфейсы

### ■ Интерфейс

Наследование ТОЛЬКО внешнего контракта

- Расширяет интерфейсы

# Объявление интерфейсов

```
interface Somethingable {  
    // константы  
    // методы  
    // вложенные классы, интерфейсы и перечисления  
}
```

- Все члены интерфейса по умолчанию обладают признаком **public**
- Применение других модификаторов приводит к ошибке компиляции
- Бывают пустые интерфейсы

# Константы в интерфейсах

```
interface Verbose {  
    int SILENT = 0;  
    int TERSE = 1;  
    int NORMAL = 2;  
    int VERBOSE = 3;  
}
```

- Имеют неявные модификаторы `public static final`
- Должны быть снабжены инициализаторами
- До Java 1.5 использовались для организации перечислений

# Методы в интерфейсах

```
interface Verbose {  
    void setVerbosity(int level);  
    int getVerbosity();  
}
```

- Имеют неявные модификаторы  
`public abstract`
- Не могут иметь модификаторов  
`native synchronized  
strictfp static final`
- Начиная с **Java 1.8** могут иметь модификатор  
`static` или `default`

# Статические методы (Java 1.8)

- Полностью аналогичны статическим методам классов
- Имеют модификатор `static`
- Содержат реализацию прямо в интерфейсе
- Вызываются **только** по имени интерфейса
- Позволяют определить методы и семантически привязать их к интерфейсу, а не к классам его реализующим
- Как правило являются утилитными методами (helper methods)

# Методы с реализацией по-умолчанию (Java 1.8)

- Имеют модификатор `default`
- Содержат реализацию прямо в интерфейсе
- Класс, реализующий интерфейс с таким методом может его переопределить (как в случае наследования)
- Позволяют добавлять новый функционал в интерфейс **без необходимости** внесения изменений в классы, использующие поздние версии этого интерфейса
- Никто не мешает предоставлять пустую реализацию...

```
public default void method() {};
```

# Расширение интерфейсов интерфейсами

```
interface NewVerbose extends Verbose, Runnable {  
    // ...  
}
```

- Интерфейс «наследует» все элементы расширяемого интерфейса
- Допускается сокрытие констант и статических методов
- Переопределение метода не несет семантической нагрузки
- Совпадение имен наследуемых методов не несет семантической нагрузки
- Однако **default**-методы могут быть переопределены, в том числе и сделаны полностью абстрактными (интерфейс-потомок содержит метод с такой же сигнатурой, но без модификатора **default**)

# Реализация интерфейсов классами

```
class MyNewThread
    extends MyThread
    implements Runnable, Verbose {
    public void run() {
        //do something
    }
}
```

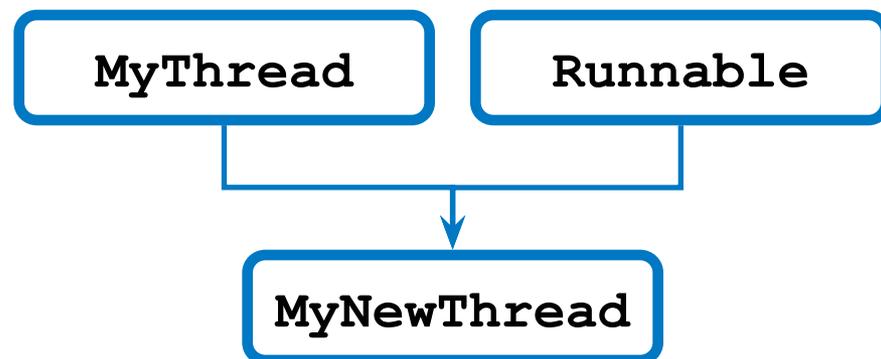
- Интерфейсы реализуются классами
- Класс может реализовывать несколько интерфейсов
- Если класс не реализует все методы «наследуемых» интерфейсов, он является абстрактным
- В случае реализации нескольких интерфейсов с одинаковой сигнатурой **default** методов, класс должен предоставить свою реализацию этого метода

# Ссылки интерфейсных типов

- Допускаются ссылки интерфейсных типов
- Такая ссылка позволяет выполнять над объектом операции, описанные во внешнем контракте, обусловленном типом интерфейса
- Такое средство существенно расширяет возможности полиморфизма

# Использование типов

- Ссылочные типы
- Неявное приведение
- Явное приведение



```
MyNewThread mnt = new MyNewThread();
```

```
MyThread mt = mnt;
```

```
Runnable r1 = mnt;
```

```
Runnable r2 = mt; // Ошибка!!!
```

```
mnt = (MyNewThread)mt; // Возможен выброс исключения
```

```
mnt = (MyNewThread)r1; // ClassCastException
```

# Интерфейс или абстрактный класс?

- Интерфейсы обеспечивают инструментарий **множественного наследования**, производный класс способен наследовать одновременно несколько интерфейсов
- Класс может расширять **единственный базовый класс**, даже если тот содержит только абстрактные методы

# Интерфейс или абстрактный класс?

- Абстрактный класс частично может быть реализован, он вправе содержать члены, помеченные как `protected` и/или `static` и т.п.
- Структура интерфейса до Java 1.8 ограничена объявлениями `public`-констант и `public`-методов без какой бы то ни было реализации
- Начиная с **Java 1.8** интерфейсы также могут быть частично реализованы благодаря статическим методам и методам с реализацией по умолчанию

# Исключительные ситуации

© Составление, Гаврилов А.В., Будаев Д.С., Стефанов М.А. 2016

Лекция 2.2

Самара  
2020

# План лекции

- Возникновение ошибок и подходы к их обработке
- Исключения и их классификация
- Объявляемые исключения
- Отлов исключений
- Выбрасывание исключений
- Создание типов исключений
- Подходы к отладке приложений

# Традиционные проблемы

- В процессе выполнения программные приложения встречаются с ситуациями, приводящими к возникновению ошибок
  - Ошибки бывают различной степени тяжести (фатальные, не фатальные, предупреждения)
  - Ошибки необходимо каким-либо способом учитывать и обрабатывать
- Ошибки возникают в случае:
  - некорректного ввода данных
  - сбоя оборудования
  - нарушения ограничений среды
  - выполнения программного кода

# Обработка ошибок

- Обеспечение стабильности и надежности работы программы
- Дружественное поведение конечного программного продукта **Противоречие!**
- Безопасность в процессе выполнения
- Удобство при написании программного кода

# Подходы к обработке ошибок

- Возвращение методом кода ошибки
  - Возвращается только код ошибки

```
int errNum = firstMethod();
if (errNum == -1) {
    // обработка 1-ой ошибки
}
else if (errNum == -2) {
    // обработка 2-ой ошибки
}
```

- Используются «свободные» значения возвращаемого типа

```
if ((ans = sqrt(val)) < 0) {
    // Обработка ошибки
}
else {
    // Продолжение вычислений
}
```

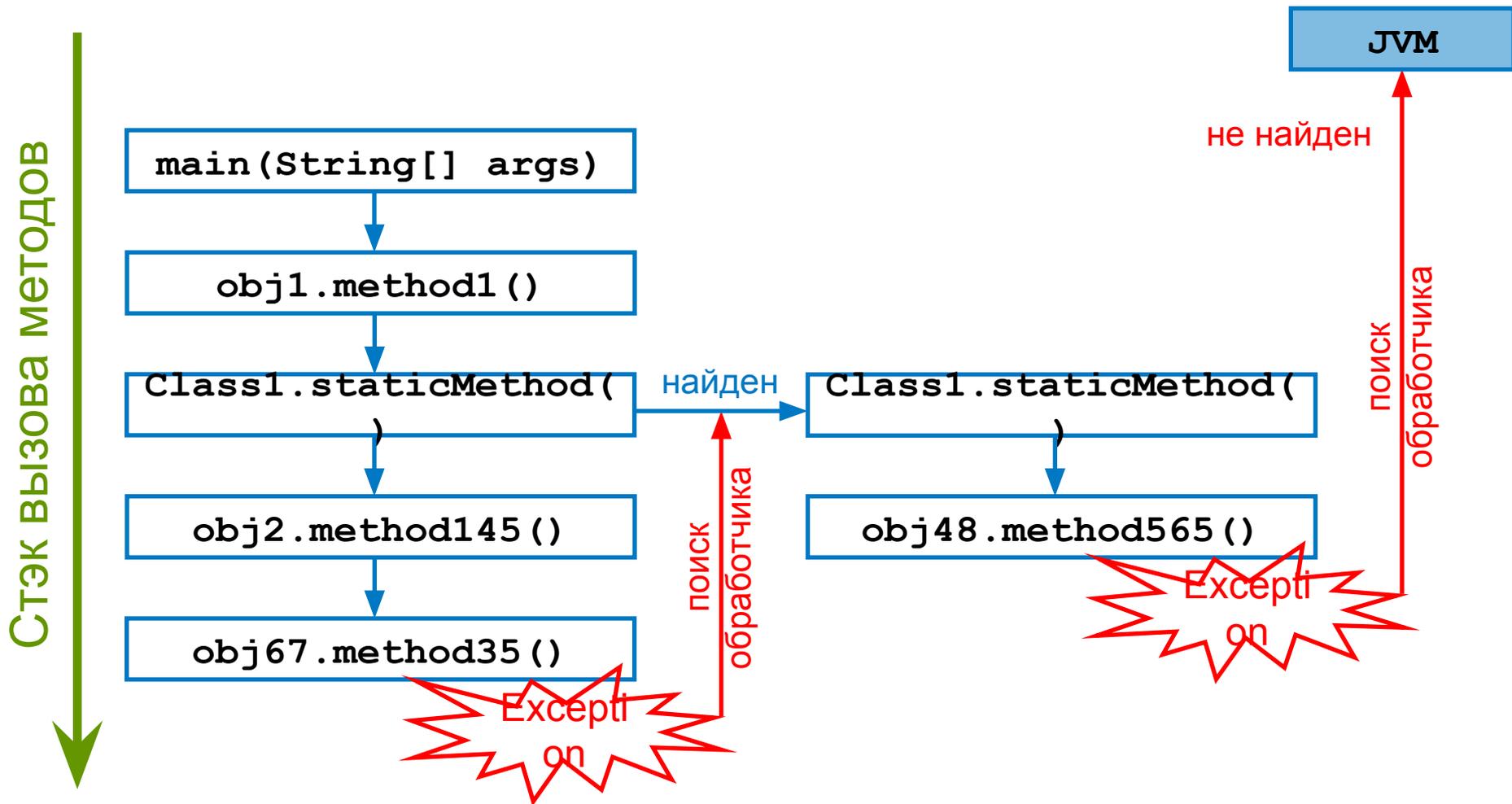
- Встроенный в язык механизм проверки и обработки

```
try {
    someBusinessLogic();
    ...
    anotherBusinessLogic()
}
catch (Exception1 e1) {
    // обработка 1-ой ошибки
}
...
catch (ExceptionN eN) {
    // обработка N-ой ошибки
}
finally {
    // выполнение завершающих
    // работу действий
}
```

# Механизм обработки

- Создается и «выбрасывается» **объект исключения**, содержащий информацию об ошибке
- **Выполнение** текущего потока вычислений **приостанавливается**
- Завершается выполнение блоков и методов в цепочке вызовов вплоть до кода, отлавливающего исключение
- Поток вычислений **возобновляется**, причем выполняется код обработчика исключения

# Поиск обработчика исключения



# Классификация исключений

## Объявляемые

(проверяемые, `checked`)

- Носят предсказуемый характер
- Указываются в объявлении метода
- Наследуют от класса `Exception`

## Не объявляемые

(непроверяемые, `unchecked`)

- Обусловлены логикой кода
- Не указываются в объявлении метода
- Наследуют от классов `RuntimeException`, `Error`

# Классификация исключений

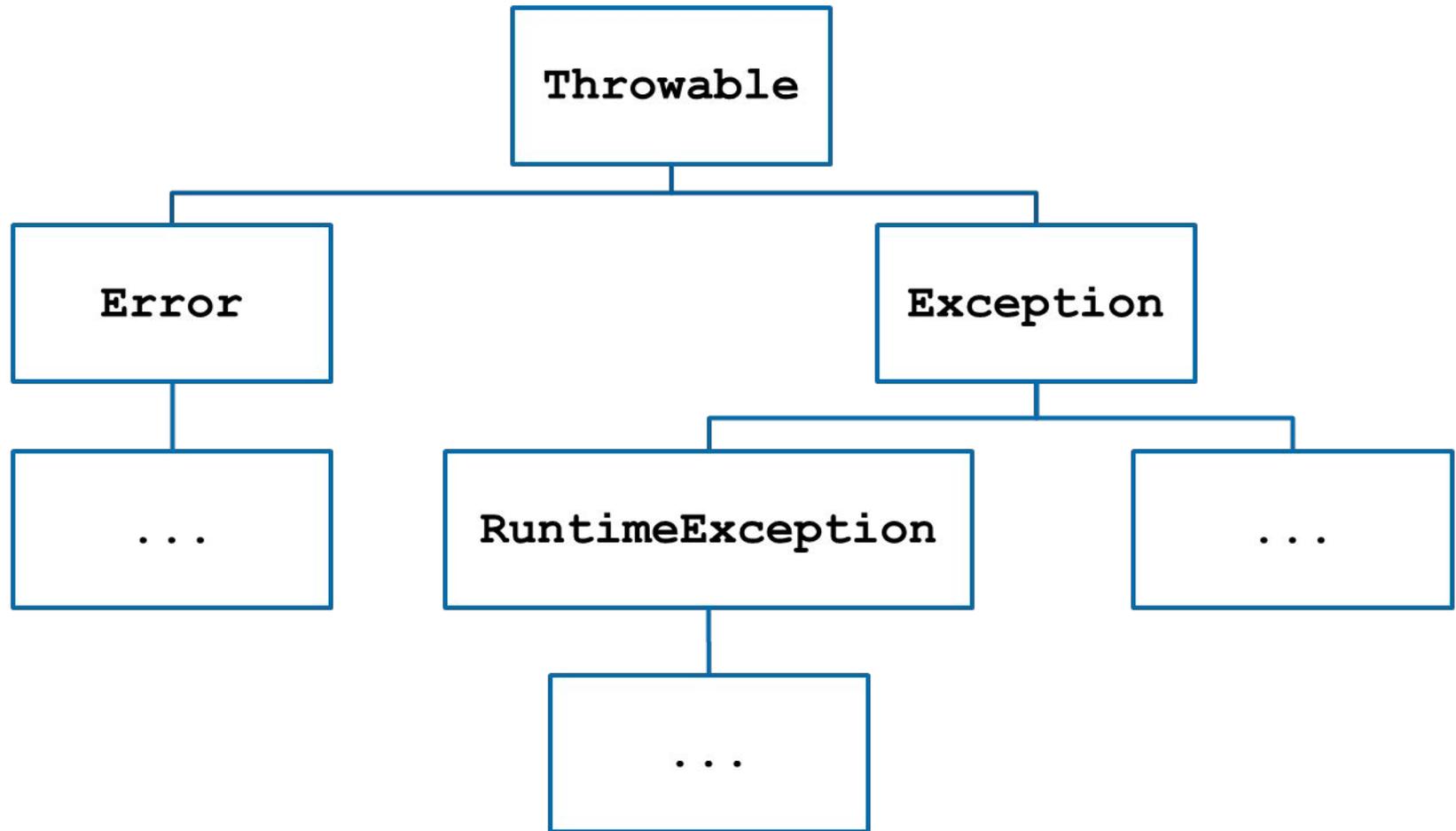
## Синхронные

- Непосредственный итог выполнения определенной инструкции
- Могут быть объявляемыми и необъявляемыми

## Асинхронные

- Не зависят от выполняемой инструкции
- Внутренние ошибки JVM
- Результат работы **deprecated** методов

# Базовые классы исключений



# Объявление исключений

- Сведения об исключениях метода не менее важны, чем тип возвращаемого им значения
- Мораль: их надо обозначать в заголовке

```
class OurClass
{
    public int someMethod() throws
        SomeException1, SomeException2
    {
        /* Код который может породить
           SomeException1 или SomeException2 */
    }
}
```

# Особенности объявления исключений

- В списке должны присутствовать те объявляемые исключения, которые не обрабатываются в теле самого метода
- Метод вправе выбросить исключение типа, наследного от заявленного в **throws**
- Запрещено генерировать **объявляемые** исключения типов, не заявленных в **throws**

# Особенности объявления исключений

- Объявляются все объявляемые исключения, не обработанные в теле метода
- Статические блоки инициализации и инициализирующие выражения не могут выбрасывать объявляемые исключения
- Нестатические блоки инициализации могут генерировать объявляемые исключения, только если их тип указан во всех **throws** всех конструкторов класса

# Вызов метода со списком исключений

## Варианты действий

- Отловить исключения и обработать их
- Отловить исключения и вместо них сгенерировать исключения типов, указанных в собственном предложении **throws**
- Объявить соответствующие исключения в предложении **throws** текущего метода и позволить им «пройти через код»

# Важное замечание

- Каждое исключение имеет как **формальную** причину возникновения, так и **фактическую**
- Исключение должно отлавливаться и обрабатываться на том уровне (по стеку, порядку вызова методов), где его:
  - можно обработать;
  - **имеет смысл** обрабатывать.
- Обработка исключений не сводится к выводу сообщений в консоль и записи в журнал (logger)!

# Отлов исключений

- Синтаксис описания обработчиков исключений
- Конструкция `try/catch/finally`

```
try {  
    Инструкции  
} catch (ТипИсключения1 идентификатор1) {  
    Инструкции  
} catch (ТипИсключения2 идентификатор2) {  
    Инструкции  
} catch (ТипИскл3 | ТипИскл4 |...| ТипИсклN идентификатор3 {  
    //такая форма появилась в версии Java 1.7  
    Инструкции  
    ...  
} finally {  
    Инструкции  
}
```

# Блок try

- Заключает в себе блок кода, выполняемый успешно при нормальных обстоятельствах
- Тело выполняется вплоть до:
  - Момента возникновения исключительной ситуации
  - Благополучного достижения конца блока
- Конкретный блок в процессе выполнения может выбросить только одно исключение

# Блок catch

- «Внутренний метод» с параметром типа исключения, которое им обрабатывается
- Способен:
  - Выполнить некоторые восстановительные действия
  - Выбросить собственное исключение
  - Осуществить необходимые действия и передать управление последующим инструкциям
- Количество блоков **catch** не регламентировано

# Блок `catch`

- Предложения `catch` рассматриваются последовательно до обнаружения среди них того, тип которого допускает присвоение выброшенного исключения
- Использовать сразу широкий тип (например, `Exception`) в качестве отлавливаемого – не лучшая мысль!
- Обработчики следует располагать по мере расширения типа исключения
- Список предложений `catch` просматривается только один раз!

# Multi catch (Java 1.7)

- Используется, если обработчики разных исключений выполняют одни и те же действия
- Но при этом сами исключения нужно различать

```
catch (ТипИсключения1 | ТипИсключения2 | ... идентификатор) {  
    Инструкции  
}
```

- Типы исключений такой формы блока catch не должны быть связаны иерархией наследования
- Одному и тому же блоку try могут соответствовать разные catch (классика и новые)

# Блок `finally`

- Блок `finally` выполняется в любом случае:
  - При успешном выполнении `try`
  - При выбрасывании исключения
  - При передаче управления по `break` или `return!`
- Блок `finally` необязателен
- Если есть `finally`, блоки `catch` необязательны
- Если присутствует, то выполняется после завершения работы остальных фрагментов кода `try`

# Блок try-with-resources (Java 1.7)

---

- Будет рассмотрен позднее

# Выбрасывание исключений

- Объявляемые и необъявляемые исключения, выбрасываемые вызываемыми **методами** и **операторами**
- Явно (принудительно) выбрасываемые исключения

```
throw referenceToThrowableObject;
```

```
throw new NoSuchElementException(name);
```

# Простейший пример

```
...  
private int cost;  
  
public void setCost (int cost) {  
    if (cost < 0)  
        throw new IllegalArgumentException("cost can't be negative");  
    else  
        this.cost = cost;  
}  
...
```

# Создание типов исключений

- Создается новый тип, наследующий от более широкого типа, **подходящего по смыслу** (например, `java.lang.IndexOutOfBoundsException`)
- Само то, что выбрасывается исключение более узкого типа, **несет в себе информацию**
- В состав нового типа могут вводиться новые поля и методы
- Чаще всего класс содержит только два конструктора (по умолчанию и с параметром-строкой), просто вызывающие конструкторы родительского класса
- Современные среды разработки облегчают создание собственных классов исключений

# Пример определения пользовательских исключений

```
class FirstException extends Exception { }

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

# Пример метода, выбрасывающего несколько объявляемых исключений

```
class ExceptionThrower {
    static public void throwException(String exceptionName)
        throws FirstException, SecondException{
        if ("first".equalsIgnoreCase(exceptionName))
            throw new FirstException();
        else
            throw new SecondException();
    }
    static public void throwSuperException(String exceptionName)
        throws Exception{
        if ("first".equalsIgnoreCase(exceptionName))
            throw new FirstException();
        else
            throw new SecondException();
    }
}
```

# Пример обработки исключений

```
static public void catchException(String exceptionName) {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (FirstException e) {
        System.err.println("first exception have been caught");
    } catch (SecondException e) {
        System.err.println("second exception have been caught");
    }
}

static public void multiCatchException (String exceptionName) {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (FirstException | SecondException e) {e.printStackTrace();}
}

static public void awfulCatchException (String exceptionName) {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (Exception e) { e.printStackTrace(); }
}
```

# Пример проброса исключений

```
static public void passException(String exceptionName)
    throws SecondException, FirstException {
    ExceptionThrower.throwException(exceptionName);
}

static public void passSuperException(String exceptionName)
    throws Exception {
    ExceptionThrower.throwException(exceptionName);
}

static public void passExceptionMore(String exceptionName)
    throws SecondException, FirstException {
    //Ошибка компиляции
    ExceptionThrower.throwSuperException(exceptionName);
}
```

# Пример генерации новых исключений

Оборачивание пойманного исключения в новое

```
static void wrapException () throws SecondException {
    try {
        ExceptionThrower.throwException("first");
    } catch (Exception e) {
        throw new SecondException(e.getLocalizedMessage(), e);
    }
}
```

“Перегенерация” пойманного исключения

```
static void rethrowException (String exceptionName) throws Exception {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (Exception e) {
        throw e;
    }
}
```

# Проблема выброса отловленных исключений

- Часто возникают ситуации, когда после обработки отловленных исключений необходимо «выбросить» их заново
- Соответственно, в предложении **throws** в сигнатуре метода необходимо отразить все эти исключения

```
static void rethrowException (String exceptionName)
    throws FirstException, SecondException {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (FirstException e) {
        // do something
        throw e;
    } catch (SecondException e) {
        // do something
        throw e;
    }
}
```

# Проблема выброса отловленных исключений

- Для решения этой проблемы приходилось описывать по блоку `catch` на каждое такое исключение
- При увеличении числа исключений, выбрасываемых методом `throwException()`, придется добавлять новые блоки `catch`, раздувая код
- Маразм приобретает поистине угрожающие масштабы, если действия в блоке `catch` перед выбросом исключений – одинаковые

# Решение 1 – multi catch (Java 1.7)

Если исключения не связаны наследованием, начиная с версии Java 1.7 можно использовать multi catch

```
static void rethrowException (String exceptionName)
    throws FirstException, SecondException {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (FirstException | SecondException e) {
        // do something
        throw e;
    }
}
```

# Решение 2 – More Inclusive Type Checking (Java 1.7)

- Компилятор Java SE 1.7 (по сравнению с более ранними версиями) производит более тщательный анализ выбрасываемых из блока `catch` исключений.
- Это позволяет указывать более узкие типы выбрасываемых исключений в сигнатуре метода, чем тип ссылки на исключение, передаваемой в оператор `throw` в соответствующем блоке `catch` в теле метода .

```
static void rethrowException (String exceptionName)
    throws FirstException, SecondException {
    try {
        ExceptionThrower.throwException(exceptionName);
    } catch (Exception e) {
        //do something
        throw e; //компилятор Java 1.1-1.6 сгенерирует ошибку
                //unreported exception Exception; must be caught
                //or declared to be thrown
    }
}
```

# Правила проверки типа вновь выбрасываемого в блоке `catch` исключения (Java 1.7)

- Это исключение может быть выброшено в блоке `try`
- Предшествующие блоки `catch` не обрабатывают это исключение
- Тип вновь выбрасываемого исключения – подтип или **супертип** параметра блока `catch`

# Отладка приложений

## Собственные средства

- Добавление дополнительного кода
- Вывод данных на печать
- Вывод данных в системные журналы (logging)
- Создание дополнительных методов проверки

## Отладчики (debuggers)

- В составе JDK, в составе среды разработки (IDE), отладчики сторонних компаний
- Использование точек останова, пошаговых режимов, просмотра состояния объектов

# Преимущества от использования исключений

- Единая логика обработки ошибок
  - Обработка ошибок на любом уровне
  - Выделение и обработка категорий ошибок
- Разделение логики по обработке ошибок и бизнес-логики приложения
- Необходимость обработки объявляемых исключений
  - Возможность действий по восстановлению

Спасибо за внимание!

# Дополнительные источники

- Нимейер, Патрик. Программирование на Java / Патрик Нимейер, Дэниэл Леук; [пер. с англ. М.А. Райтмана]. – Москва : Эксмо, 2014. – 1216 с.
- Шилдт, Г. Java 7- Полное руководство - 8th Edition. – М.: ООО «И.Д. Вильямс», 2012г. – 1104 с.
- Хорстманн, К. Java 2. Библиотека профессионала. Том 1. Основы [Текст] / Кей Хорстманн, Гари Корнелл. – М. : Издательский дом «Вильямс», 2010 г. – 816 с.
- Эккель, Б. Философия Java [Текст] / Брюс Эккель. – СПб. : Питер, 2011. – 640 с.
- JavaSE at a Glance [Электронный ресурс]. – Режим доступа: <http://www.oracle.com/technetwork/java/javase/overview/index.html>, дата доступа: 14.09.2020.
- JavaSE APIs & Documentation [Электронный ресурс]. – Режим доступа: <http://docs.oracle.com/en/java/javase/14/>, дата доступа: 14.09.2020.