



Самарский государственный аэрокосмический университет
имени академика С.П. Королёва

Объектно-ориентированное программирование

Исключительные ситуации и их обработка

Занятие 4.1

© Составление,
А.В. Гаврилов, 2014
А.П. Порфирьев, 2015

Самара
2014

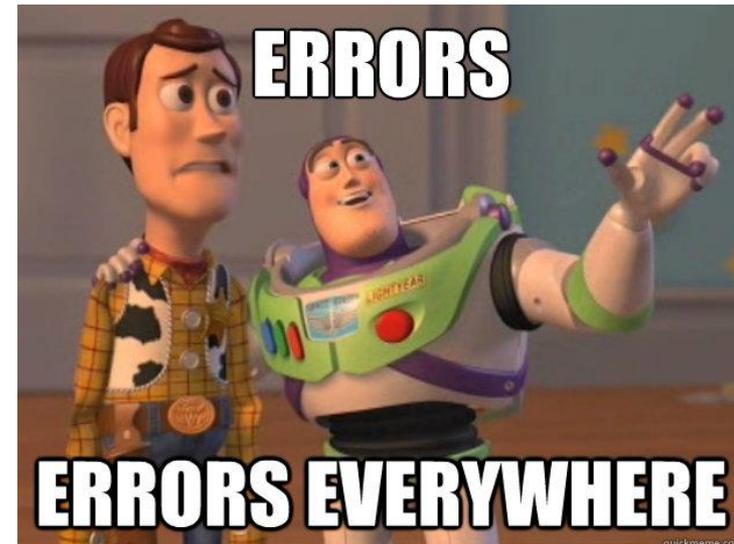
План лекции

- Возникновение ошибок и подходы к их обработке
- Механизм обработки исключений
- Классификация исключений
- Объявляемые исключения и их особенности
- Отлов исключений
- Выбрасывание исключений
- Создание типов исключений
- Отладка приложений



Э... Проблемы

- В процессе выполнения программные приложения встречаются с ситуациями, приводящими к возникновению ошибок
 - Ошибки бывают различной степени тяжести
 - 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
- Безопасность в процессе выполнения
- Удобство при написании программного кода

Противоречие!



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
}
```

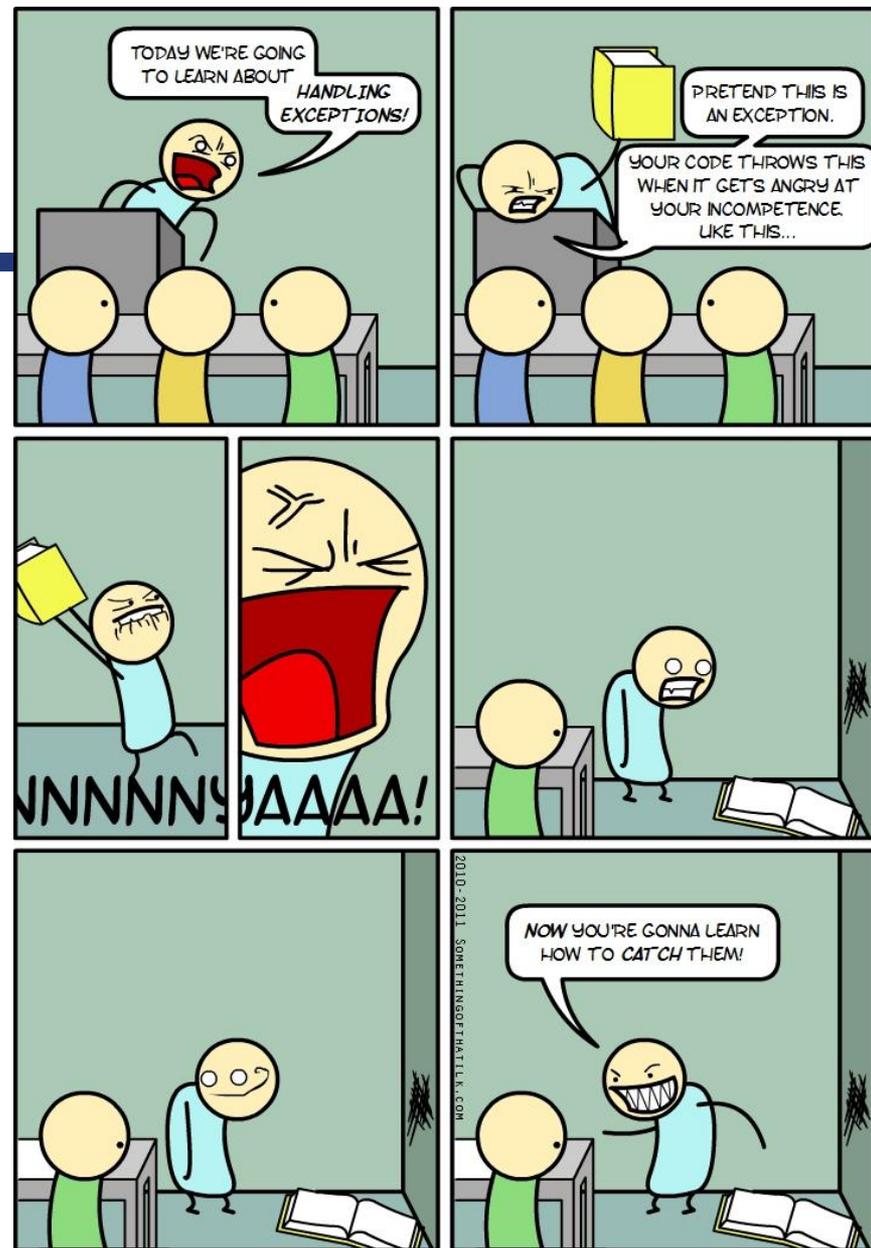
- Встроенный в язык механизм проверки и обработки

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

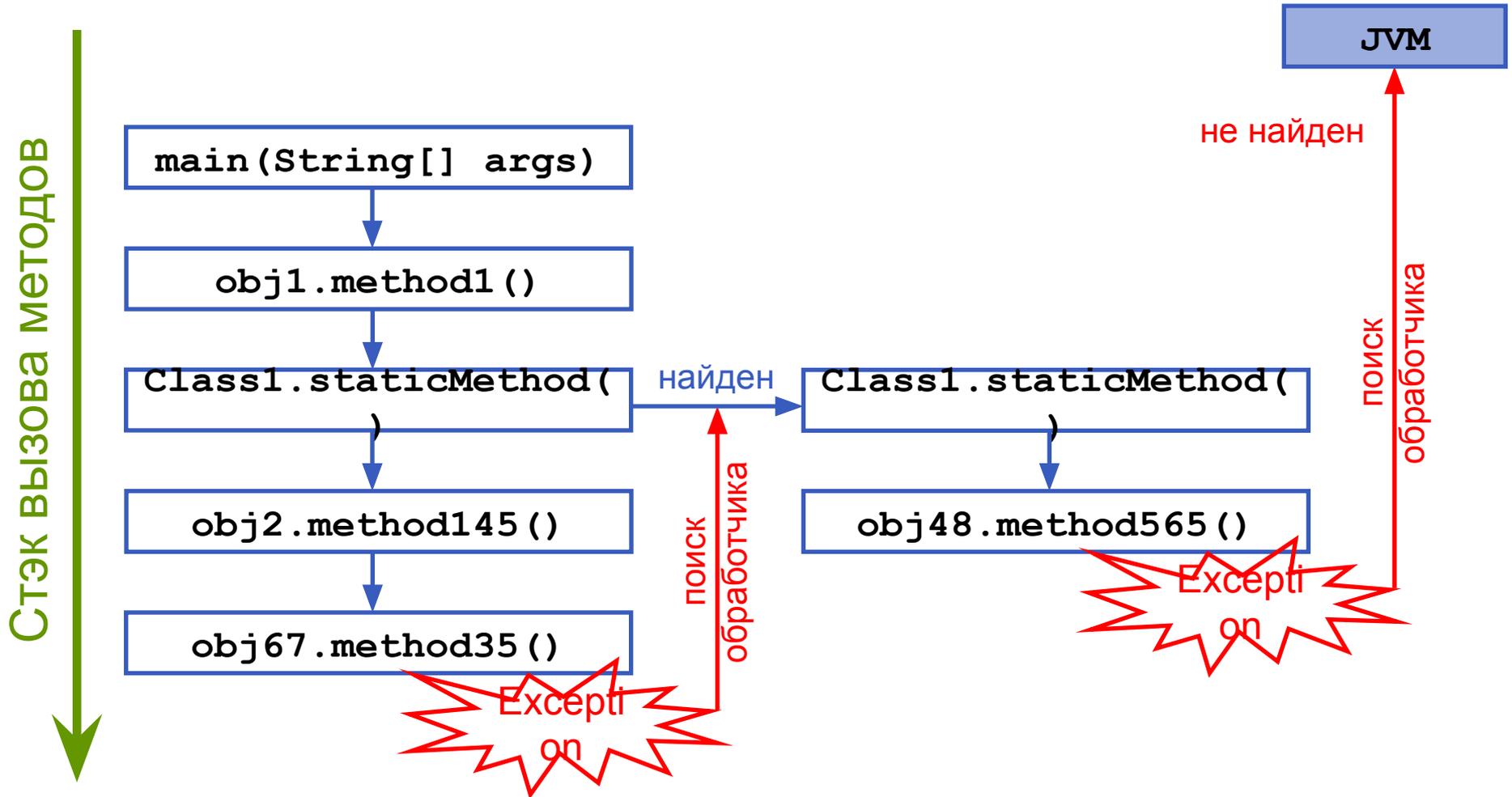


Механизм обработки

- Создается и «выбрасывается» объект исключения, содержащий информацию об ошибке
- Выполнение текущего потока вычислений приостанавливается
- Завершается выполнение блоков и методов в цепочке вызовов, пока не будет найден код, отлавливающий исключение
- Выполнение потока 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`

Необъявляемые

(непроверяемые / `unchecked`)

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



Классификация исключений

Синхронные

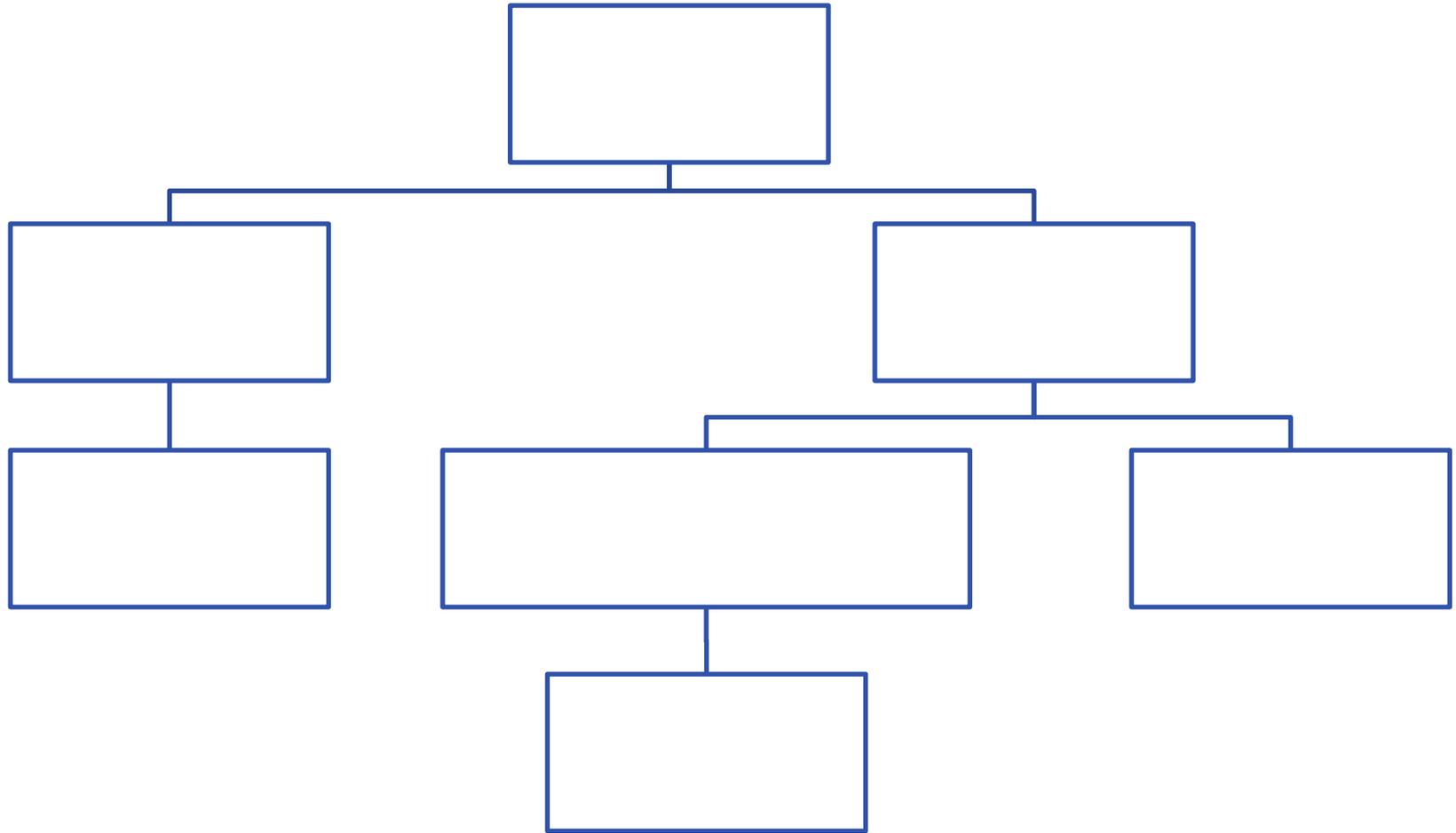
- Непосредственный итог выполнения определенной инструкции
- Могут быть объявляемыми и необъявляемыми

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
- Они являются частью контракта операции и контракта класса
- Мораль: их надо обозначать в заголовке

```
class OurClass {  
    public int someMethod() throws Exception1, Exception2 {  
        /* Код, который может породить  
        Exception1 или Exception2 */  
    }  
}
```



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 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» вашего метода



Важные замечания

- 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.
- Выбрасывание исключения – это реакция на нарушение внешнего контракта
- Обработка исключений не сводится к выводу сообщений в консоль и записи в журнал (logger)!



Отлов исключений

- Особый синтаксис описания обработчиков исключений
- Конструкция `try/catch/finally`

```
try {  
    Инструкции  
} catch (ТипИсключения1 идентификатор1) {  
    Инструкции  
} catch (ТипИсключения2 идентификатор2) {  
    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 широкий тип (например, **Exception**) в качестве отлавливаемого – не лучшая мысль!
- Список предложений **catch** просматривается только один раз!



Отлов исключений нескольких типов (Java 7)

- Нужные типы исключений разделяются вертикальной чертой |
- Блоков `catch` с таким синтаксисом может быть много
- В блоке `catch` можно использовать только общие методы указанных типов исключений
- Переменная, куда помещается ссылка на объект исключения, неявно приобретает модификатор `final`



Отлов исключений нескольких типов (Java 7)

- Хотелось бы:

```
try {  
    // Что-то ввод данных и рефлексия  
}  
catch (ClassNotFoundException | IOException ex) {  
    logger.log(ex);  
    throw ex;  
}
```



Блок `finally`

- Блок `finally` необязателен
- Если присутствует, то выполняется после завершения работы остальных фрагментов кода `try`
- Блок `finally` выполняется в любом случае:
 - Успешное выполнение
 - С выбрасыванием исключения
 - С передачей управления по `break` или `return`



Выбрасывание исключений

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

```
throw referenceToThrowableObject;
```

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

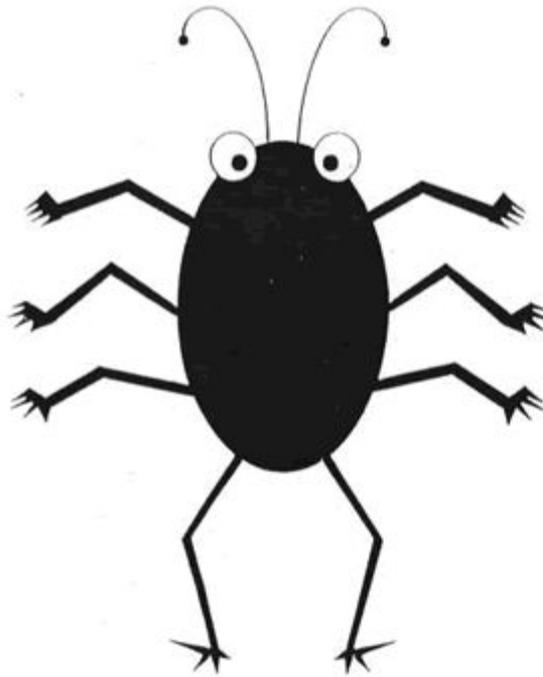


Создание типов исключений

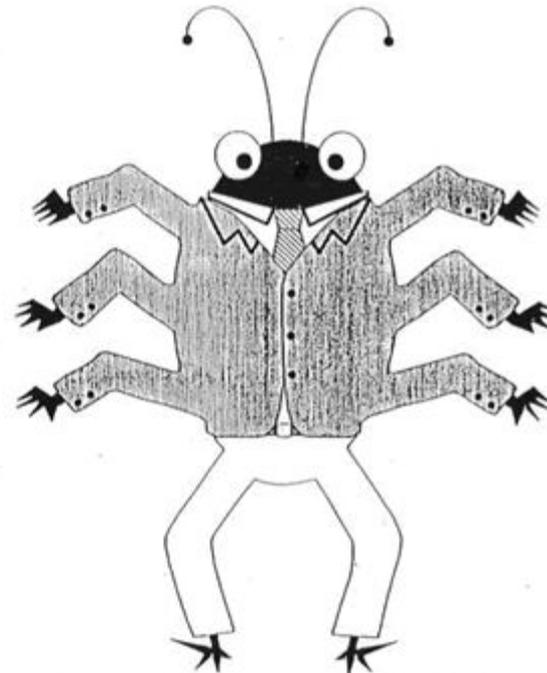
- Создается новый тип, наследующий от более широкого типа, **подходящего по смыслу** (например, `java.lang.IndexOutOfBoundsException`)
- Само то, что выбрасывается исключение более узкого типа, **несет в себе информацию**
- В состав нового типа могут вводиться новые поля и методы
- Чаще всего класс содержит только 2 конструктора (по умолчанию и с параметром-строкой), просто вызывающие конструкторы родительского класса
- Современные среды разработки облегчают создание собственных классов исключений



Отладка приложений



BUG



FEATURE



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





Самарский государственный аэрокосмический университет
имени академика С.П. Королёва

Объектно-ориентированное программирование

Наследование и его особенности

Занятие 4.2

© Составление,
А.В. Гаврилов, 2014
А.П. Порфирьев, 2015

Самара
2014

План лекции

- Виды наследования
- Расширение классов
- Переопределение методов
- Соккрытие полей
- Завершенные и абстрактные методы и классы
- Интерфейсы и их описание
- Использование интерфейсов



Наследование в Java

Виды наследования

■ Класс

Наследование как типа,
так и реализации

- Расширяет класс
- Реализует интерфейсы

■ Интерфейс

Наследование
ТОЛЬКО типа

- Расширяет интерфейсы



Расширение классов

- Класс может расширить **только один класс**

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

- Если родительский класс не указан явно, родительским становится класс **java.lang.Object**



Конструкторы дочерних классов

- Вызываются для создания объектов дочерних классов
- Могут вызывать друг друга по ключевому слову `this ()`
- В первой строчке должны вызывать конструкторы базового класса по ключевому слову `super ()`
- Ключевое слово `super ()` может не использоваться, только если в родительском классе существует конструктор по умолчанию



Конструкторы дочерних классов. Пример

```
class ClassA {  
    public int a;  
  
    public ClassA(int a) {  
        this.a = a;  
    }  
}  
  
class ClassB extends ClassA{  
  
}
```



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



Забавный пример

```
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

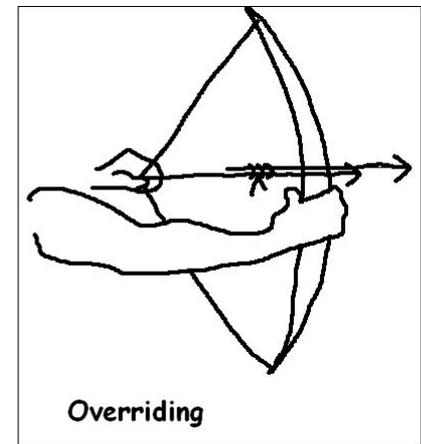
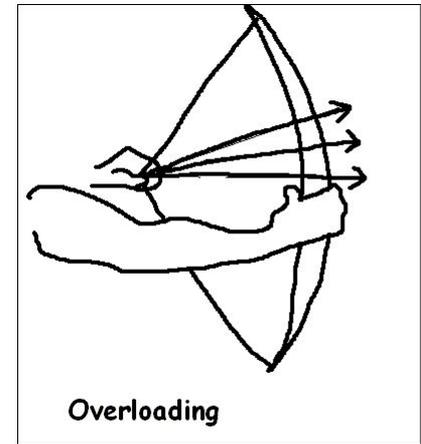
```
public static void main(String[] args) {  
    ExtendShow ext = new ExtendShow();  
    SuperShow sup = ext;  
    ext.show();  
    sup.show();  
    System.out.println("ext.str = " + ext.str);  
    System.out.println("sup.str = " + sup.str);  
}
```

```
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** при переопределении не может сужаться
- Методы **private** не переопределяются



Переопределение методов

- В предложении **throws** дочернего метода не может быть типов исключений, не совместимых 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
- 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
 - Вызов конструктора родительского класса `super(...)`
 - Вызов метода родительского класса `super.method(...)`
 - Обращение к полю родительского класса `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



Давешний 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
```



Замечание

Важно понимать, что:

- Переопределение методов – фундаментальный механизм, позволяющий изменить реализацию класса без изменения типа
- Соккрытие полей – следствие отсутствия ограничений на имена полей



Завершенные методы и классы

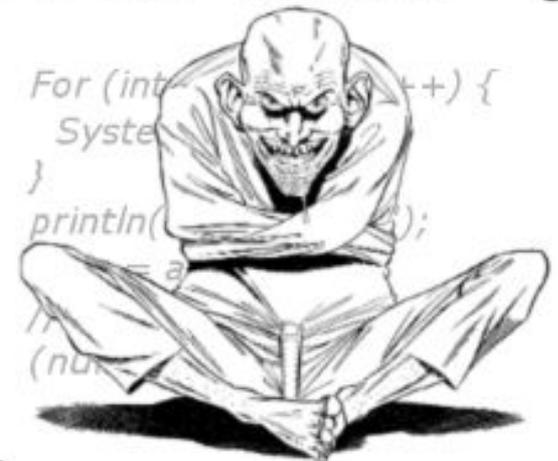
- Завершенный метод не допускает переопределения

```
final public int getValue() { ... }
```

- Завершенный класс не допускает расширения

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

Final Class {



}



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!**



Понятие интерфейса

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



Объявление интерфейсов

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

- Все члены интерфейса по умолчанию обладают признаком **public**
- Применение других модификаторов редко имеет смысл
- 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



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`



Расширение интерфейсов интерфейсами

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

- Допускается сокрытие констант
- Переопределение метода не несет семантической нагрузки
- Совпадение имен наследуемых методов не несет семантической нагрузки



Реализация интерфейсов классами

```
class MyNewThread
    extends MyThread
    implements Runnable, Verbose {
    ...
}
```

- Интерфейсы реализуются классами
- Класс может реализовывать несколько интерфейсов
- Если класс не реализует все методы «наследуемых» интерфейсов, он является абстрактным



Интерфейс или абстрактный класс?

- Интерфейсы обеспечивают инструментарий безопасного **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 ограничена объявлениями `public`-констант и `public`-методов без какой бы то ни было реализации



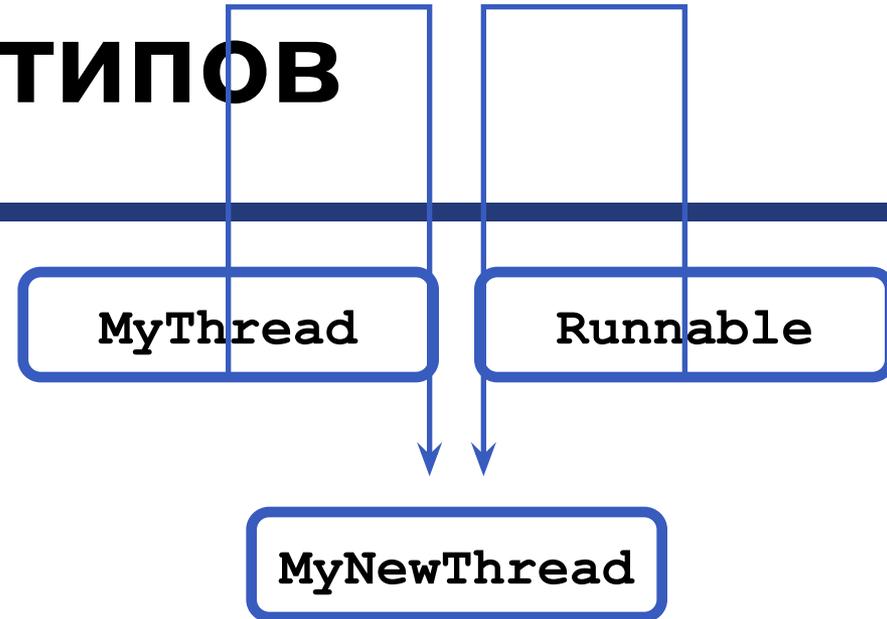
Ссылки интерфейсных типов

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



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
```



Пустые интерфейсы

- Существуют пустые интерфейсы, объявления которых не содержат ни констант, ни методов
- Реализация таких интерфейсов обычно означает способность объекта к чему-либо
- Ссылка такого типа редко имеет смысл (т.к. внешний контракт пуст)
- Даже такая ссылка позволяет выполнять методы объекта...

а именно методы, объявленные в классе **Object**, поскольку они есть у абсолютно любого объекта



Спасибо за внимание!

Дополнительные источники

- Арнолд, К. Язык программирования Java [Текст] / Кен Арнолд, Джеймс Гослинг, Дэвид Холмс. – М. : Издательский дом «Вильямс», 2001. – 624 с.
- Вязовик, Н.А. Программирование на Java. Курс лекций [Текст] / Н.А. Вязовик. – М. : Интернет-университет информационных технологий, 2003. – 592 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: 21.10.2011.
- JavaSE APIs & Documentation [Электронный ресурс]. – Режим доступа: <http://www.oracle.com/technetwork/java/javase/documentation/api-jsp-136079.html>, дата доступа: 21.10.2011.

