Программирование на Java
(2022 - 2023)

Лекция 6

Обработка ошибок и исключительных ситуаций
Assertions
Использование информации о типах
Reflection

к.т.н. Гринкруг Е.М. (email: egrinkrug@hse.ru)

«Плохо написанная программа не должна запускаться»…

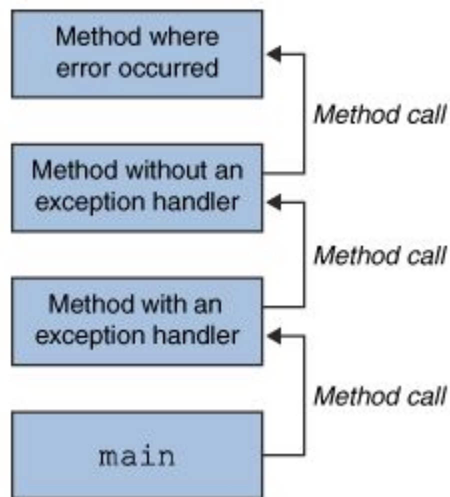
- ***Верно ли это (как иногда считают)?***
  - **Что такое «хорошо написанная программа» и/или что такое просто «хорошая программа»?**
- **В идеале: чем раньше обнаружена ошибка, тем лучше.**
- **Обнаружить ошибку можно (исключая этап «написания»):**
  - **На этапе компиляции (перед запуском программы);**
  - **На этапе загрузки классов на выполнение (может оказаться, что не все нужные классы имеются или могут сосуществовать вместе);**
  - **На этапе тестирования (при запуске программы в целях тестирования);**
  - **На этапе выполнения...**
- **В Java всегда работает динамический контроль исполнения программы.**
- **Система обработки ошибок и средства восстановления после ошибок – важные факторы надежности кода *и «живучести программы».***
- **Надежная система должна строиться из надежных компонент.**

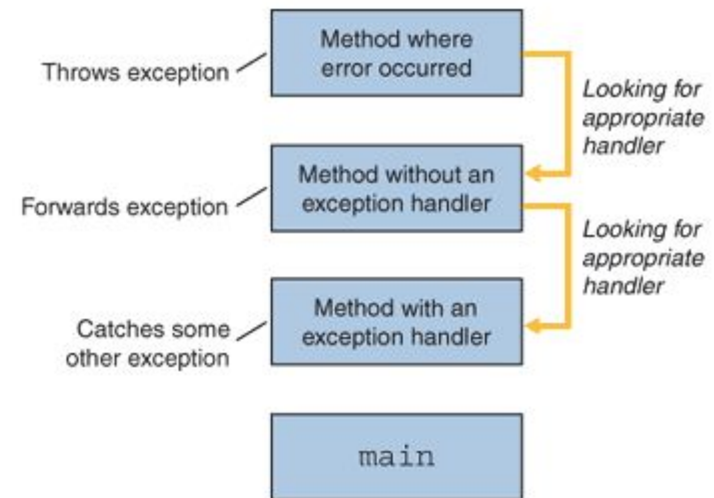
- В С-подобных языках схемы обработки ошибок устанавливались «соглашениями», а не являлись частью языка. Имеется долгая история развития средств реакции на ошибки и их обработки (проверки кода завершения метода, программные прерывания, и др.)

Exceptions – отклонения от (исключения из) «нормального» поведения

- **Exception – событие, происходящее во время выполнения программы и делающее её «нормальное» продолжение невозможным, но, возможно, позволяющее обеспечить переходом для продолжения «обходной путь».**
- **«*Исключительная ситуация*» (Exception) отличатся от «обычной» ошибки, при которой в текущем контексте есть достаточно информации для преодоления затруднений. При исключительной ситуации в текущем контексте нет возможности обработки, и требуется выйти из текущего контекста, передав проблему на более высокий уровень.**
- **Когда внутри метода (блока) возникает исключительная ситуация, метод создает объект, представляющий исключение (exception object), и отдает его runtime-системе. Объект исключение создается «в куче», как и другие объекты, и содержит информацию об ошибочной ситуации и состоянии программы. Создание этого объекта и передача его системе называется «возбуждением исключения» (throwing an exception, «выкидывание исключения»).**

- Система пытается найти то место в программе (блок кода его обработчика), которое обработает исключение, – т.н. exception handler;
- «Кандидаты» ищутся в стеке вызовов, начиная с текущего метода. Подходящий находится, если тип объекта исключения приводим к типу, заявленному обработчиком (т.е. «ловится» им); иначе – выполнение завершается. На картинке ниже стек «растет» вверх, как его показывает, например, отладчик в Idea:





Аргументы исключения

- Исключения – это объекты, создаваемые с помощью оператора **new**, который выделяет память (из кучи) и вызывает конструктор;
- Все стандартные исключения имеют три конструктора:
  - Стандартный (без параметров);
  - С аргументом типа String (для описания исключения);
  - С аргументом типа Throwable (причина исключения).
- Корнем иерархии наследования всех типов исключений является класс **`java.lang.Throwable`** (см. API) – базовый класс для всех объектов, которых можно «выкинуть» с помощью **throw**.
- Ключевое слово **throw** обеспечивает выход из текущего блока или метода. Этот выход отличается от того, который используется при нормальном выполнении. Например:

```
if (t == null) throw new NullPointerException (“t happened to be null”);
```

Блок try { }

- Если внутри метода инициируется исключение (или это делает другой вызванный там метод), данный метод завершает работу оператором **throw**, если только не перехватывать исключения блоком **try** и обрабатывать обработчиками исключений **catch**:

```
try {
    // код, способный возбуждать исключения
} catch ( ThrowableType1 id1) {
    // обработка исключений первого типа...
} catch ( ThrowableType2 id2) {
    // обработка исключений второго типа...
} // ...
```

Блок catch ( ) { }

- Каждый обработчик исключений **catch** *(англ. - ловушка)* похож на метод с одним аргументом заданного типа, который может быть использован внутри обработчика;
- Обработчики всегда следуют прямо за блоком **try {}** ;
- После выполнения найденного по типу исключения нужного блока **catch** поиск обработчиков-ловушек заканчивается;

- ***Поиск нужной ловушки по типу исключения аналогичен поиску нужного перегруженного метода с одним аргументом: ищется тот, кто «точнее / ближе» соответствует типу исключения / параметра.***

Ловля более одного исключения в одном обработчике

- Начиная с Java 7, один **catch**() – блок может ловить более одного типа исключений, что сокращает код;
- Типы отлавливаемых исключений при этом перечисляются через «или» (что кодируется символом ‘|’), например:

```
…    catch( IOException | SQLException ex ) {
         logger.log(ex);
         throw ex;
     }
```

В этих случаях параметр ex в **catch**() {…} - неявно является final (и ему ничего нельзя присвоить внутри блока **catch**( ) {…})

Блок finally { }

- Блок **finally** { } – если он есть – выполняется всегда после выхода из блока **try** { }. Пишется за блоками catch (при их наличии). Блок **finally** { } обязательно выполняется (даже при непредвиденных исключениях).
- Блок **finally** { } полезен и вне связи с исключениями – для гарантии выполнения завершающего (cleanup) кода: это хороший стиль даже если не предвидится обработка исключений - он выполнится всегда *(если не прервется сам – остановив JVM или сорвавшись по exception…)*
- Возможно использование **try { } finally { }** без блоков **catch ( ) { }**.
- Полезно разделять **try** / **catch** и **try** / **finally**:

```
try{
        try{
        // код, который может генерировать exception…
        } finally { /* close file … */}
} catch ( IOException e ) { … }
```

Особенности работы блока finally{ }

- Если в блоке **finally** есть **return**, результат выполнения может быть «неожиданным»:
  - **Если в середине try{ ... } мы выходим по return, то будет выполнен блок finally {… }. При выходе по return из finally { ... } может измениться возвращаемое значение. Что выдает следующий метод?:**

```
public static int f (int n) {
        try {
                return   n * n;
        } finally {
                return  0;
        }
}
```

- Лучше избегать исключений в методах, помещаемых внутри блока finally {…} *(Почему?)*. Хорошее обсуждение исключений см. в Effective Java (by J.Bloch)

The try-with-resources Statement (Java 7 и далее...)

- Это **try** – блок, который декларирует ***ресурсы***, требующие закрытия после употребления;
- Такой блок гарантирует, что после него все его ресурсы будут закрыты; в сущности – это синтаксический приём (избавление от **finally {}**)
- В качестве ***ресурса*** выступает любой объект, который реализует интерфейс **`java.lang.AutoCloseable`** (что включает все объекты, реализующие интерфейс **`java.io.Closeable`**);
- Например:

```
static String readFirstLineFromFile (String path) throws IOException {
 try (BufferedReader br = new BufferedReader(new FileReader (path))) {
       return br.readLine();
 }
}
```

... Раньше, до Java 7...

- Вместо ***try-with-resource*** использовали блок **finally** { }, например:

```
static String readFirstLineFromFile (String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
      return br.readLine();
    } finally {
      if (br != null) br.close();
    }
}
```

- Имеются различия в случае, когда и readLine() и close() кидают exception (различие – в том, какое из них «подавляется» -> надо «подыграть» и проверить...Использовать это надо аккуратно…).

Требование: «лови или специфицируй»

- Код, который может возбуждать некоторые исключения, обязан быть заключен в один из следующих контекстов выполнения:
  - **Оператор try { }, который ловит такое исключение**, либо
  - **Метод, который указывает, что он выдает такое исключение.**

Иначе, ***для некоторых типов исключений, которые называются контролируемыми (checked)***, код не будет скомпилирован (не пройдет проверку компилятором еще до выполнения)***.***

Такое требование называется ***Catch Or Specify***
(лови или специфицируй, «лови или кидай»,
«лови сам или предупреждай, чтобы ловили другие»...)

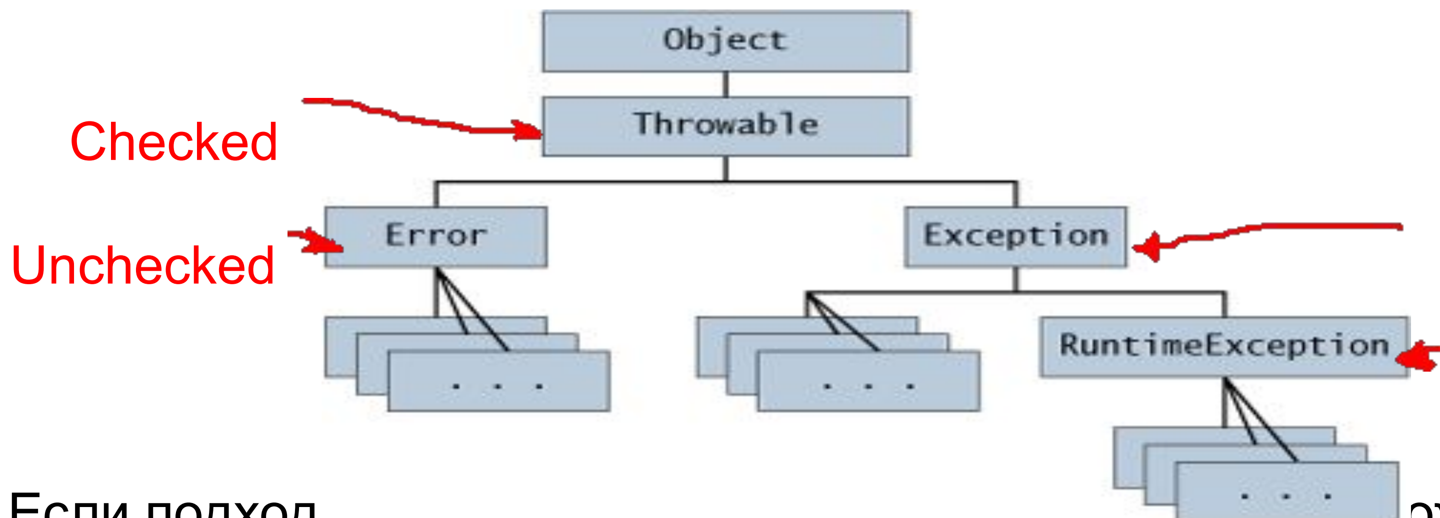
Классификация исключений

- **Checked exceptions** – исключения, которые *проверяются и обработка которых навязывается еще на этапе компиляции программы* - они обязаны соответствовать требованию *Catch Or Specify*. Пример: **`java.io.FileNotFoundException`**

- **Unchecked exceptions** – *не проверяются на этапе компиляции* и не подпадают под требование *Catch Or Specify* :
  - **Error** – неустранимая ошибка, которую нельзя обойти программным образом, но надо выдать диагностику. Пример: **`java.io.IOError`**, **`java.lang.ClassFormatError`** и др. *(какие вы уже видели?)*
  - **RuntimeException** – ошибка, связанная с некорректным поведением самой программы приложения. Пример: **`java.lang.NullPointerException`**.

Иерархия (наследования) **исключений**

- **Throwable** *(то, что может «выскочить»)* имеет двух прямых потомков: **Error** и **Exception** (то есть **Error – не есть Exception**):



- Если подходящих стандартных типов исключений нет, можно создать свой класс исключения. Он может быть подклассом **Exception** (или его потомка), хотя может наследоваться и из **Throwable** или **Error** (в чем смысла мало)…

Классификация исключений - примеры

- **Error – иерархия (unchecked / не контролируемые компилятором):**
  - Внутренние ошибки, контролируемые средой исполнения и/или нехватка ресурсов
    - **Продолжать «нормальную» работу программы при них затруднительно (нельзя):**
      - Неверный формат класс-файла;
      - Переполнение стека
      - Другие...
- **Exception – иерархия**:
  - **RuntimeException – иерархия (unchecked / не контролируемые компилятором)** :
    - **Возникают в результате ошибок программирования**, таких как:
      - Неверное приведение типов (cast);
      - Выход за пределы массива;
      - Обращение к неинициализированной переменной (по ссылке null);
      - Другие...
  - **Все остальные (checked / контролируемые компилятором) :**
    - **Возникают при корректной работе, но при непредвиденных обстоятельствах**:
      - Загрузчик классов не нашел нужный класс;
      - Нет поддержки клонирования;
      - Другие...(от файловой системы, сетевых программ, и пр.)

java.lang
<<interface>>
Serializable
checked
Throwable
unchecked
Error
checked
Exception
VirtualMachineError
ThreadDeath
UnknownError
StackOverflowError
OutOfMemoryError
RuntimeException
unchecked
ReflectiveOperationException
java.sql
SQLException
java.io
IOException
java.util
NoSuchElementException
IndexOutOfBoundsException
ArithmeticException
ClassCastException
NullPointerException
IllegalArgumentException
NoSuchFieldException
IllegalAccessException
ClassNotFoundException
NoSuchMethodException
EOFException
FileNotFoundException

Объявление контролируемых (checked) исключений

- Объявление о том, что метод может генерировать исключения:

```
public Image loadImage(String s) throws EOFException, MalformedURLException {…}
```

- Метод подкласса не может генерировать более общие контролируемые исключения, чем замещаемый им метод суперкласса *(исключения могут только конкретизироваться или не возникать вовсе).*
- Если метод суперкласса вообще не генерирует контролируемые исключения, то и переопределенный метод в подклассе этого сделать не может.
- Если метод объявляет, что может генерировать контролируемые исключения определенного класса, то он может генерировать также исключения его подклассов *(«если ловим рыбу, то – и селедку словим...» или «если кидаемся рыбой, то можем и селедкой кидаться...», - и все потому, что селедка – это рыба*).

Цепочки исключений (Chained exceptions) – *добавлены by Josh Bloch в JDK 4*

- Часто бывает так, что одно исключение является причиной другого.
- Полезно об этом знать...
- Следующие методы и конструкторы класса Throwable работают с chained exceptions:
  - **Throwable getCause() - дает причину данного throwable;**
  - **Throwable initCause(Throwable) – задает причину данного throwable;**
  - **Throwable (String, Throwable) и Throwable (Throwable) – создают новый throwable по указанному как параметр;**
- Например:

```
try { /* some code here…  */ } catch (IOException e)
{ throw new SampleException (“Other IOException”, e)}
```

Использование Throwable для анализа стека

- Трассировка стека – список вызовов методов до данной точки программы – полезный механизм отладки;
- Доступен в классе Throwable (т.е. - для всех исключений);
- До JDK1.4 применялся метод printStackTrace() - в консоль;
- Начиная с JDK 4, есть удобные методы программного доступа

  StackTraceElement [ ] getStackTrace();
- StackTraceElement имеет методы, позволяющие получить имя файла, номер строки кода, имя класса и метода – информацию, полезную для отладки и отображаемую методом toString().
- Соответствующий API совершенствовался в последующих JDK (см.) – в частности, в JDK 9 с внедрением средств модульности в JVM (о них – позже…).

Рекомендации по работе с исключениями

- Обработка исключений не заменяет необходимости простой проверки; обработка исключения значительно уступает ей в производительности *(попробуйте оценить - измерить разницу)*;
- Не надо плодить блоки **try** { }, отдельные для каждой операции;
- Используйте иерархию исключений, не перехватывайте сразу все (**catch**(Throwable th)) и не ограничивайтесь только RuntimeException.
- Не затыкайте исключения: …**catch** (Exception e) { /*«наглухо»*/}
- Бывает лучше сгенерировать нужное исключение, чем вернуть null и разбираться с ним в дальнейшем;
- Не обрабатывайте все исключения, делегируйте часть «наверх», указывая, что метод, например, throws IOException ...
- Для Java 8(+) есть нюансы при работе с потоками (Java 8 streams…). Обсудим, когда до них дойдем…

Общие архитектурные соображения

- В последнее время, с внедрением в Java средств функционального программирования, изменяется и взгляд на exceptions handling…
- Например, подход PFJ (**P**ragmatic **F**unctional **J**ava, краткий обзор которого приведен тут (https://dzone.com/articles/introduction-to-pragmatic-functional-java) основан на двух правилах:
  - **A**void **n**ull **A**s **M**uch **A**s **P**ossible (**ANAMAP** rule);
  - **N**o **B**usiness **E**xceptions (**NBE** Rule).
- Ставшие традиционными для скалярных стековых архитектур **(за ~30(+) лет)** идеи обработки «исключительных ситуаций», которые были заложены в аппаратуре стековых архитектур **(например, в архитектуре «Эльбрус-2(+)»)**, в JVM и в списанном с нее MS dotNet (~ 27(+) и 22(+) лет назад, соответственно) привели к тем механизмам, которые мы сейчас рассматриваем. В Java эти механизмы слегка совершенствовались **(от JDK1.0 до JDK7 и далее)**.

- Эти идеи соответствуют только скалярным последовательным вычислениям.
- Они являются средством, позволяющим простым пользователям-программистам сравнительно удобно реагировать, в частности, на аппаратно-программные прерывания, связанные с контролем скалярных последовательных вычислений. Они сводят обработку таких прерываний (обычно отлавливаемых ядром операционной системы) к обслуживанию соответствующих «ловушек исключительных ситуаций» в стеке вызовов методов (процедур).
- Динамические средства (операционная система и runtime-поддержка языка) обеспечивают такую схему обработки exceptions при скалярных последовательных вычислениях, при этом напрочь ломая весь (скалярный) конвейер выполнения инструкций. Отсюда – большие издержки производительности вычислений на аппаратном-программном уровне: обработка прерываний, поиск нужной ловушки по стеку и вызовы соответствующих методов реакции.

- В некоторых компьютерах, не таких убогих, как наш стандартный «ширпотреб», имеются аппаратные сигналы по переполнению в арифметике (и/или по «потере значимости»), по делению на 0 не только на целочисленных, но и на других форматах, сигналы, формируемые по разным другим (параллельно контролируемым) условиям выполнения вычислений.
- Для параллелизма данных, при «функциональных» вычислениях (в которых функция не является скалярной, а обрабатывает наборы/потоки данных), при вычислениях в стиле SIMD, такой подход не годится. Там гораздо лучше работает логика «маскирования»: вычисления продолжаются для всех данных параллельно, но их обработка учитывает (векторные(!)) признаки результатов векторных вычислений, влияющие на их результат. Это идеологически согласуется с новшествами вычислений с «условными результатами» (Optional, Optional.map() и flatMap()), которые сходны с «вычислениями под маской» и «уплотнением под маской», реализованными в машинах М.А.Карцева (40 и/или 50 лет назад - в разных машинах – аппаратно (!)).

- Мне повезло иметь возможность программировать для таких машин – много (20 лет) и в разных областях (от ядра ОС до компиляторов...)
- Приятно видеть, что сейчас в JDK17 появляются средства организации векторных вычислений под масками, идеологически повторяющие средства таких векторных машин. Они хорошо согласуются с функциональным подходом к программированию: работают сразу «со всеми значениями» функции, а не поочередно-последовательно с каждым из них (скалярно).
- Поэтому: какая машина (в том числе – виртуальная машина), такие и средства программирования на ней (в том числе – средства обработки исключений).
- Для того, чтобы дойти до понимания и использования таких вещей, надо прикладывать труд (системного программиста)…

Assertions

- Ключевое слово assert (*англ. - утверждение*) используется для логического контроля выполнения программы. Появилось с JDK 1.4. Употребляется в двух форматах:

assert <booleanExpression>;

если **<booleanExpression>** дает **false**, возникает **AssertionError**.

Не рекомендуется использовать это для проверки аргументов public-методов: для этого есть **IllegalArgumentException**;

assert <booleanExpression> : <errorMessageExpression>;

- Тут **errorMessageExpression** не м.б. вызовом метода с типом возвращаемого значения **void**

- Тут можно указать дополнительную информацию про assertion. Эта информация будет получена с помощью метода toString() из любого объекта, который может быть выдан как результат вызова метода с параметрами…

Преимущества использования java assertions

- Полезны для проверки данных (data validation) – в динамике;
- Облегчают отладку сложных программ;
- При надлежащем использовании assertions ошибка в программе или данных может проявляться ближе к ее источнику;
- Дополняют другие средства тестирования (JUnit assertions) возможностями проверки исполнения программы с реальными данными (в реальных, а не тестовых условиях);
- Дисциплинируют программиста и улучшают качество кода;
- Добавляют уверенности в правильной работе программы – в стиле:
  - 5 минут – «полет нормальный» и т.д. (т.н. «логический контроль поведения»)
- Упрощают refactoring (переделку) кода при переходе на новую версию, например:
  - assert <oldMethodVersionResult> == <newOneResult>;

Замечания по использованию Java Assertions

- Assertions разрешаются / запрещаются опциями запуска Java VM:

  -enableassertions / **-disableassertions**

  -ea / -da

  или с точностью до указанных в опции пакетов (с подпакетами), например:

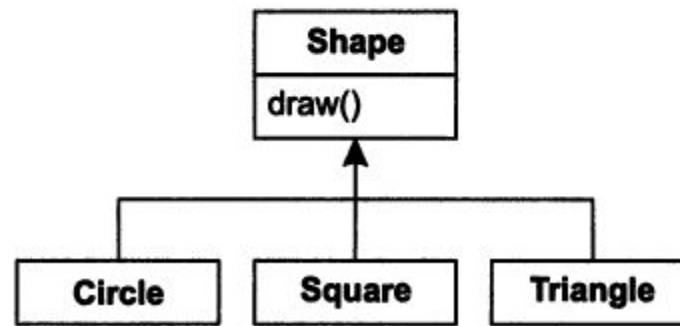
  -ea:myPackageName…

  По умолчанию (при запуске Java VM) – assertions не разрешены (disabled by default)

- Java Assertions не заменяют Exceptions, а дополняют их;
  - Не надо использовать assertions для проверки параметров public-методов (для этого есть **IllegalArgumentException**);
- Java Assertions не заменяют JUnit-assertions, а дополняют их.

Runtime Type Information

- Механизмы получения и использования информации о типах ***во время исполнения программы*** **(at Runtime)**– важные средства объектно-ориентированного программирования. На них, в частности, основаны многие важные инструменты программирования и отладки (например: как Idea показывает содержимое в «закладке» Structure ?)…
- Рассмотрим пример использования такой информации о типах:
  - Абстрактный класс Shape с методом draw();
  - Три конкретных производных класса (могут быть и другие...):



```java
package com.grinkrug.rtti;

abstract class Shape {
    void draw(){System.out.println(this + ".draw()");}
    public abstract String toString();
}
class Circle extends Shape {
    public String toString(){return "Circle";}
}
class Square extends Shape {
    public String toString(){return "Square";}
}
class Triangle extends Shape {
    public String toString(){return "Triangle";}
}
public class Shapes {
    public static void main(String[] args){
        Shape[] myShapes = {new Circle(), new Square(), new Triangle()};
        for(Shape shape : myShapes){
            shape.draw();
        }
    }
}
```

Как это работает? Что будет выведено?

- **Сам класс Shape не может инстанциироваться: он абстрактный;**
- **Метод draw() базового класса Shape неявно использует метод toString() для вывода «названия» контретной фигуры;**
- **Для этого методу System.out.println() передается ссылка this;**
- **Метод toString() в Shape – абстрактный. Он:**
  - **Переопределяет метод toString() из java.lang.Object;**
  - **Обязывает производные от Shape классы определить toString().**
- **Когда this встречается в выражении конкатенации строк, автоматически вызывается его toString();**
- **При помещении экземпляров конкретных форм (shape) в массив абстрактных форм (shapes) происходит восходящее преобразование типа; массив хранит просто объекты Shape.**
- **Далее работает полиморфизм: для каждой фигуры вызывается ее метод draw() и ее метод toString();**
- **Выводится результат:**

```
Circle.draw()
Square.draw()
Triangle.draw()
```

- ***Как быть, если все-таки хочется узнать, с каким точно типом Shape мы работаем (например, чтобы покрасить треугольники в красный цвет)?***

Как узнать конкретный тип объекта, имея ссылку базового типа?

Методы класса Class для анализа типа, представленного классом

По объекту Class можно узнать практически все, что может потребоваться знать о типе.

- **Является ли данный класс интерфейсом?**
  `public boolean isInterface ( );`
- **Является ли данный класс классом массива?**
  `public boolean isArray ( );` - и если является, то: `public Class getComponentType ( );`
- **Является ли данный класс Enum-ом?**
  `public boolean isEnum ( );`
- **Представляет ли данный класс примитивный тип?**
  `public boolean isPrimitive ( );`
- **Является ли данный класс анонимным?**
  `public boolean isAnonimousClass ( );`
- **Является ли данный класс локальным?**
  `public boolean isLocalClass();`
- **Является ли данный класс классом-членом (вложенным)**
  `public boolean isMemberClass ( );`
- **В каком классе декларирован данный класс?**
  `public Class getDeclaringClass ( );`

- **Какие интерфейсы реализуются данным классом или интерфейсом?**
  public Class[ ] getInterfaces();
- **Какой суперкласс у данного класса?**
  public Class getSuperclass();
- **Представляет ли данный класс суперкласс / суперинтерфейс другого?**
  public boolean isAssignableFrom (Class another);
- **Каким загрузчиком классов был получен данный класс?**
  public ClassLoader getClassLaoder();
- **Какому пакету принадлежит данный класс?**
  public Package getPackage();
- **Является ли переданный объект инстансом данного класса?**
  public boolean isInstance(Object o);

это – динамический эквивалент оператора instanceof:

```
if (shape instanceof Triangle ) {
        Triangle t = (Triangle) shape;
}
```

Приведите пример, когда нужен такой динамический эквивалент ? Чем по сути отличаются этот метод **isInstance()** *и оператор* **instanceof** *?*

Reflection – работа с классом в динамике

- Как быть, если мы хотим работать в такой ситуации:
  - Мы пишем программу, которая работает с некоторыми классами;
  - Нам передали какие-то классы (без исходного кода), и сказали: используйте их.
- Эта ситуация – типичная для компонентного программирования:
  - Кто-то делает классы (компоненты) разного назначения;
  - Мы используем их «по мере поступления», как детали при **конструировании** нашего изделия, без «переплавки», как есть (т.е. без перекомпиляции вообще);
- Подобная компонентность может быть полезна и на уровне межкомпьютерного (сетевого) взаимодействия:
  - Для удаленных вызовов методов (Remote Method Invocation, RMI), где мы можем работать с классами (на другой машине), про существование которых мы не знали при компиляции нашей программы.
  - Для универсальной сериализации / десериализации объектов, и. др.
  - *Как бы вы подошли к организации такой работы с классами - компонентами?*

ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

Класс Class и reflection

- Класс Class сам поддерживает концепцию Reflection *(как в переводе?)*
- Дополнительная поддержка дана в пакете **`java.lang.reflect`**
- В библиотеке **`java.lang.reflect`** есть классы, объекты которых создаются JVM для представления членов класса в динамике выполнения программы. Основными из них являются:
  - Constructor;
  - Method;
  - Field.
- Класс Class имеет методы, позволяющие получить эти объекты для заданного экземпляра класса и использовать их в динамике. Кроме того, в классе Class есть методы для анализа его вложенных классов, и т.д., что позволяет узнать про его содержимое в подробностях…

Извлечение информации о конструкторах класса

(*продолжение обзора методов класса Class*)

- Получение всех public конструкторов класса:
  public Constructor [ ] getConstructors ( ) throws SecurityException;
- Получение всех конструкторов класса (включая приватные):
  public Constructor [ ] getDeclaredConstructors ( )
  throws SecurityException;
- Получение заданного public - конструктора класса:
  public Constructor getConstructor (Class… paramTypes)
  throws NoSuchMethodException, SecurityException;
- Получение заданного конструктора, декларированного в классе:
  public Constructor getDeclaredConstructor (Class… paramTypes)
  throws NoSuchMethodException, SecurityException;

Инстанциирование класса

- **Имеющийся класс сам может создавать свои инстансы**
  - специальным методом (при наличии конструктора без параметров), который устарел:

@Deprecated (since = “9”):

public Object newInstance() throws InstantiationException,IllegalAccessException

  - с использованием объектов – конструкторов
- **Объект конструктор (java.lang.reflect.Constructor)** позволяет:
  - выполнить дальнейший анализ (узнать все про данный конструктор), т.е. получить типы параметров, исключений, ...
  - выполнить создание и инициализацию нового объекта класса, которому принадлежит конструктор (класса, где конструктор был декларирован), с заданными параметрами; например:

```
static Object createInstance (Constructor c, Object[ ] params) throws Exception {
        return c.newInstance(params);
}
```

Извлечение информации о методах класса

(*продолжение обзора методов класса Class*)

- Получение всех public методов класса (включая унаследованные):

  public Method[ ] getMethods ( ) throws SecurityException;
- Получение всех методов класса (включая приватные, но исключая все унаследованные):

  public Method[ ] getDeclaredMethods ( ) throws SecurityException;
- Получение заданного public - метода класса:

  public Method getMethod (String name, Class… paramTypes) throws NoSuchMethodException, SecurityException;
- Получение заданного метода, декларированного в классе:

  public Method getDeclaredMethod (String name, Class… paramTypes) throws NoSuchMethodException, SecurityException;

Использование объекта Method

- Как и объект Constructor, объект Method может показать информацию о себе (типы параметров, возвращаемого значения, принимает ли он переменное число аргументов, и пр.)
- Основное применение – определенная классом Method операция вызова метода:

```
public Object invoke (Object obj, Object … args)  throws IllegalAccessException,
AllegalArgumentException, InvocationTargetException;
```

- Например, в своей программе вы можете написать:

```
static Object callMethod (Object o, Method m, Object…params) throws Exception  {
return m.invoke (o, params);
}
```

- При вызове статического метода объект не нужен (и м.б. задан null)…

Работа с полями класса

(*продолжение обзора методов класса Class*)

- Получение всех public полей класса (включая унаследованные):

  public Field[ ] getFields ( ) throws SecurityException;

- Получение всех полей класса (включая приватные, но исключая все унаследованные):

  public Field[ ] getDeclaredFields ( ) throws SecurityException;

- Получение заданного public - поля класса по имени:

  public Field getField (String name)
  throws NoSuchFieldException, SecurityException;

- Получение заданного поля, декларированного в данном классе:

  public Field getDeclaredField (String name)
  throws NoSuchFieldException, SecurityException;

Работа с полями класса

- Класс Field используется для доступа к значению поля у конкретного объекта;
- Если поле в классе – статическое, то конкретный объект в операциях доступа игнорируется и может быть null;
- Для доступа к значению поля в классе Field есть набор методов

чтения:

```
public Object get (Object concreteObject)
        throws IllegalArgumentException, IllegalAccessExpubliction;
public int getInt (Object concreteObject)
        throws IllegalArgumentException, IllegalAccessExpubliction;
// и  - аналогично - для других примитивных типов;
```

и записи:

```
public void set (Object concreteObject, Object newValue)
        throws  IllegalArgumentException, IllegalAccessExpubliction;
public void setInt (Object concreteObject, int newValue)
        throws IllegalArgumentException, IllegalAccessExpubliction;
// и – аналогично - для других примитивных типов
```

Reflection и массивы

- **В библиотеке reflection есть утилитный класс для динамического создания массивов и доступа к ним – java.lang.reflect.Array.**
- **В этом классе:**
  - **Операции создания массива:**

```
public static Object newInstance(Class componentType, int length)
    throws NegativeArraySizeException;
public static Object newInstance(Class componentType, int… dimensions)
    throws IllegalArgumentException, NegativeArraySizeException;
```

  - **Операции доступ к элементам массива:**

```
public static Object get (Object array, int index)
    throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
public static void set (Object array, int index, Object newValue)
    throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
```

    и их аналоги для примитивных типов элементов массива...
- ***При каких ситуациях полезна динамическая работа с массивами?***

Замечания о динамических средствах reflection

- Методов довольно много:
  - они добавляются по мере развития языка и его runtime-поддержки.
  - надо смотреть/знать/понимать API.

- У них есть недостатки, и они – обратная сторона достоинств:
  - Нет статического контроля, все откладывается на runtime;
  - Нет защиты (без специальных динамических средств) от обращений к приватным полям и методам (постепенно меняется в новых JDK…);
  - Динамические вызовы происходят медленнее (хотя этот аспект реализации постоянно оптимизируется)

Усовершенствования reflection

- С внедрением новых средств в JDK 5(+) появились соответствующие дополнения в reflection-библиотеке и в самом классе Class;
- Они касаются новых способов определения и работы с типами, добавленными в Java (generic types)
  - Мы рассмотрим такие усовершенствования и средства reflection для них далее;
- Некоторые операции, связанные с reflection, позволяли обходить защиту («насильно» разрешая доступ к приватным членам класса).
- В JDK 9(+) это слегка изменилось:
  - **Стали учитываться модули, (мы это потом обсудим специально);**
  - **Пакет с типом, подвергающимся рефлекции, должен быть открыт модулю, использующему setAccessible().** *Это – не решение проблемы, но «ограничение»...*
- Все контролировалось SecurityManager'ом, которого по умолчанию нет и которого скоро как-то уберут: он @Deprecated, начиная с JDK17…

Класс java.lang.reflect.Proxy

- Мы отмечали, что развитие программирования – в целом – связано с повышением роли динамики:
  - Более развитые аппаратно-программные средства всегда обладают большей динамикой... *(BTW: ваши примеры?...)*
- Одним из примером такого развития в Java является внедрение механизма Dynamic Proxy (начиная с JDK 1.3).
- Используя этот механизм, можно в динамике (at runtime) создавать новые классы, которые реализуют заданный набор интерфейсов;
  - *(BTW: А какими вообще способами можно получить класс, реализующий заданный интерфейс?)*
- Этот механизм нужен в тех случаях, когда во время компиляции вы не знаете, какие интерфейсы вам надо реализовать...
  - *(BTW:Кто может привести пример такой ситуации?)*

- Для «прикладного» программиста такая ситуация встречается не часто; но для «системного» программиста – это важное средство...
- Предположим, что вы хотите создать объект некоторого класса, который реализует один или несколько интерфейсов, которые **могут не быть известны к моменту компиляции** ...
  - *(BTW; какими вообще способами можно создать объект некоторого класса?)*
- ***Необходимо определить новый класс во время исполнения программы !***
- Можно сгенерировать исходный текст, вызвать компилятор и потом загрузить полученный класс-файл...
  - *(BTW: где/когда возможен пример такого подхода? Встречались с ним?)*
- Можно воспользоваться библиотекой (такие библиотеки есть...) для кодогенерации at runtime…(они умеют генерировать байткод сразу);
- Это – громоздко, долго и требует распространять компилятор (или библиотеку кодогенерации) вместе с вашей программой...

Механизм [Dynamic] Proxy

- Класс Proxy может создавать новые классы во время исполнения программы;
  - Эти классы реализуют указанные вами интерфейсы;
- В частности, proxy-класс имеет следующие методы:
  - Все методы, требуемые указанными интерфейсами;
  - Все методы, определенные в классе java.lang.Object;
    - ***(BTW: кто может их сейчас перечислить и/или указать их количество?)***
- Однако, вы не можете в динамике определить новый код методов;
- Вместо него вы должны предоставить т.н. ***invocation handler*** ;
- Это объект класса, реализующего интерфейс InvocationHandler с единственным методом:

  Object invoke(Object proxy, Method method, Object[ ] args);

  - *(BTW:где/когда мы с вами уже видели похожий метод?)*

- Всякий раз, когда вызывается метод proxy-объекта, вызывается метод invoke() указанного handler'а с объектом-методом и параметрами, соответствующими исходному вызову;
- Этот handler сам разбирается, что делать дальше...
- Для создания proxy-объекта надо вызвать метод newProxyInstance() класса Proxy с параметрами:
  - classLoader (или null – по умолчанию), // *(BTW: что это и зачем нужно?)*
  - Массив Class[ ] – по одному элементу на каждый нужный интерфейс,
  - Invocation handler;
- Осталось понять:
  - Как мы определяем invocation handler? И
  - Что мы можем делать с полученным proxy-объектом...
- Ответы на эти вопросы определяются нашими задачами – тем, как мы собираемся использовать этот dynamic proxy - механизм ...

Использование [Dynamic]Proxy

- Proxy можно использовать в разных целях:
  - Для отправки вызовов методов на удаленный сервер;
  - Для связи событий GUI с действиями работающей программы
    - ***(BTW: где / когда это было бы полезно?);***
  - Для трассировки вызовов методов программы в целях отладки; ...
- Рассмотрим последний случай на следующем примере:

```java
/**
 * An invocation handler that prints out the method name and parameters, then
 * invokes the original method
 */
class TraceHandler implements InvocationHandler {
    /**
     * Constructs a TraceHandler
     * @param t the implicit parameter of the method call
     */
    public TraceHandler(Object t) {
        target = t;
    }

    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        // print implicit argument
        System.out.print(target);
        // print method name
        System.out.print("." + m.getName() + "(");
        // print explicit arguments
        if (args != null) {
            for (int i = 0; i < args.length; i++) {
                System.out.print(args[i]);
                if (i < args.length - 1) System.out.print(", ");
            }
        }
        System.out.println(")");
        // invoke actual method
        return m.invoke(target, args);
    }

    private Object target;
}
```

- Для создания proxy-объекта, который трассирует вызовы своих методов, поступаем так:

```
Object target = . . .; // кому proxy будет передавать вызовы…
// конструируем его wrapper (BTW: что такое wrapper?)
InvocationHandler handler = new TraceHandler(target);
//  конструируем proxy для одного или более интерфейсов:
Class[ ] interfaces = new Class[ ] { Comparable.class, … };
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

- Теперь, если методы из указанных интерфейсов вызовутся на объекте proxy, мы напечатаем имя метода и параметры, после чего позовем такой метод у объекта target...
- Полезно посмотреть, какой класс для proxy сгенерируется, какое у него имя и другие атрибуты...

Свойства Proxy-классов

- Будучи созданными в динамике, proxy-классы являются – тем не менее – «нормальными» JVM-классами;
- Все proxy-классы наследуются из Proxy-класса и наследуют его поле – invocation handler, определенное в суперклассе Proxy;
- Все дополнительные данные для выполнения задачи, стоящей перед proxy-объектом, должны храниться в invocation handler’e;
- Все proxy-классы переопределяют следующие методы класса java.lang.Object, просто отправляя их к invoke() invocation handler’a:
    - toString();
    - equals();
    - hashCode();

  другие методы класса Object– *(это какие?)* – не переопределяются;
- Имя proxy-класса не определяется (но можно его узнать: $Proxy);
    - **Полезно «обследовать» такой класс средствами reflection…**

Method Handles

(для продвинутых / интересующихся)

- Внедрены в JDK7 (и реализованы в пакете `java.lang.invoke`).
- «A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values»...
- Это – новый низкоуровневый механизм поиска, получения доступа и вызова методов (он появился для реализации динамических языков на JVM).
- Объекты MethodHandle – immutable и не имеют видимого состояния.
- Для создания и использования этих объектов нужно сделать 4 шага:
  - **Создание специального объекта – «ищейки» для поиска;**
  - **Создание объекта с описанием типа искомого метода;**
  - **Нахождение method handle’a;**
  - **Вызов с помощью method handle‘a.**

Method Handles и Reflection

- Method Handles внедрены для использования совместно со старым java.lang.reflect API:
  - **Они предназначены для разных целей и имеют разные характеристики.**
- С точки зрения производительности, Method Handles API может быть заметно быстрее, чем Reflection API:
  - **Все проверки доступа производятся на этапе создания, а не выполнения.**
  - **Это различие особенно заметно при наличии SecurityManager'а, так как при нем в поиске классов и их членов выполняются дополнительные проверки.**
- Однако, производительность – лишь одна сторона вопроса. Method Handles труднее использовать из-за отсутствия в них ряда средств (проверок флагов доступности, которые есть у reflections и других)...
- Тем не менее, Method Handles предоставляют ряд полезных на практике возможностей...

Создание Lookup

- Первое, что надо сделать для работы с MethodHandle – получение объекта Lookup – фабрики, отвечающей за создание MethodHandle' ов для методов, конструкторов и полей, которые м.б. видимы классу Lookup.

- С помощью MethodHandles API можно создать lookup-объект с разными режимами доступа:
  - **Создание lookup-объекта, предоставляющего доступ к public-методам:**
    MethodHandles.Lookup publicLookup = MethodHandles.publicLookup();
  - **Если мы хотим иметь доступ еще и к private и protected методам, то**
    MethodHandles.Lookup lookup = MethodHandles.lookup();

Создание MethodType-объекта

- Для создание MethodHandle’а объекту Lookup требуется определение типа метода, и оно предоставляется с помощью класса MethodType.
  - **MethodType представляет аргументы и возвращаемый тип, принимаемые и возвращаемые MethodHandle’ом (или передаваемые и ожидаемые вызывающим MethodHandle объектом).**
  - **Структура MethodType проста и состоит из возвращаемого типа вместе с соответствующим количеством типов параметров, которые должны быть правильно сопоставлены между MethodHandle’ом и всеми его вызывающими.**
- Так же, как и MethodHandle, экземпляры MethodType - immutables.
- Например, MethodType для возвращаемого типа java.util.List и массива Object’ов как типа аргумента:

```
MethodType mt = MethodType.methodType (List.class, Object[].class);
```

- MethodType, который возвращает значение типа int и принимает Object:

```
MethodType mt = MethodType.methodType ( int.class, Object.class );
```

Находим MethodHandle

- Lookup-фабрика предоставляет набор методов, которые позволяют найти подходящий MethodHandle, принимая во внимание область действия метода. Рассмотрим разные сценарии, начиная с простого.

- MethodHandle для методов:
  - **Инстанс-методы можно найти с использованием findVirtual(). Например, для метода concat() класса String мы делаем:**

```
MethodType mt = MethodType.methodType(String.class, String.class);
MethodHandle concatMH = publicLookup.findVirtual (String.class, "concat", mt);
```

- MethodHandle для статических методов:
  - **Используем findStatic() при поиске MethodHandle’а для метода asList():**

```
MethodType mt = MethodType.methodType(List.class, Object[].class);
MethodHandle asListMH = publicLookup.findStatic( Arrays.class, "asList", mt);
```

- MethodHandle для конструкторов:
  - **Используется метод findConstructor(). Для поиска конструктора Integer(String s) имеем:**

```
MethodType mt = MethodType.methodType( void.class, String.class);
MethodHandle newIntegerMH = publicLookup.findConstructor ( Integer.class, mt);
```

- MethodHandle для полей:
  - **С помощью MethodHandle можно получить доступ к полям. Имеем:**

```
class Book {
    String id;
    String title;
   // constructor
}
```

Мы можем добыть MethodHandle, который соответствует getter'у поля (если это позволяет доступность поля):

```
MethodHandle getTitleMH = lookup.findGetter (Book.class, "title", String.class);
```

- MethodHandle для приватных (!) методов можно получить, используя способ получения MethodHandle'ов из объектов- Method'ов (reflection):
  - **Пусть в классе Book есть приватный метод:**

```
class Book {
     String id;
     String title;
     private String bookInfo () { return id + “ > ” + title; }          // …
}
```

Тогда:

```
Method bookInfoMethod = Book.class.getDeclaredMethod(“bookInfo");
    formatBookMethod.setAccessible (true);
    MethodHandle bookInfoMH = lookup.unreflect(bookInfoMethod);
```

... И мы видим, что сюда переносятся недостатки контроля доступа из reflection…

Вызовы с помощью MethodHandle(s)

- Получив MethodHandle, можно использовать три способа вызовов:
  - **invoke();**
  - **invokeWithArguments();**
  - **invokeExact().**

- При использовании invoke(…) мы фиксируем набор аргументов, но разрешаем выполнять cast и boxing/unboxing аргументов и возвращаемых значений (в допустимых пределах...).
- При использовании invokeWithArguments(…) мы (дополнительно к предыдущему) разрешаем менять количество аргументов.
- При использовании invokeExact(…) мы не разрешаем никаких преобразований при передаче параметров и возвращаемых значений.

- С помощью MethodHandle(s) можно оперировать массивами и выполнять различные другие полезные операции;
- Однако, не надо этими механизмами злоупотреблять и пользоваться ими следует только тогда, когда это необходимо (нет других, более высокоуровневых средств)...
- Подробности следует смотреть в соответствующей API документации.

Источники для чтения по теме Exceptions

Основная литература:

- **https://docs.oracle.com/javase/tutorial/essential/exceptions/**
- **Хорстманн, том 1, глава 7 (рус. или англ. – есть в SmartLMS).**

Дополнительная литература:

- **J.Bloch. Effective Java (соответствующие разделы). Есть в SmartLMS**
- **https://www.oreilly.com/ideas/handling-checked-exceptions-in-java-streams?imm_mid=0f6a21&cmp=em-prog-na-na-newsltr_20170923**
- **http://www.theserverside.com/tutorial/OCPJP-Use-more-precise-rethrow-in-exceptions-Objective-Java-7**
- **https://www.wisdomjobs.com/e-university/java-exception-handling-interview-questions.html**
- **Java Notes for Professionals. Chapter 68 и 131. Есть в SmartLMS**

Для интересующихся развитием свежих идей:

- **https://dzone.com/articles/introduction-to-pragmatic-functional-java**
- **https://dev.to/siy/we-should-write-java-code-differently-210b**

Полезные источники по теме Reflection и MethodHandle(s)

- Учебники Учебники (Учебники (Хорстман и др.Учебники (Хорстман и др.)
- Java Reflection Tutorial (Java Reflection Tutorial (см. Java Reflection Tutorial (см. SmartLMS Java Reflection Tutorial (см. SmartLMS в папке Java Reflection Tutorial (см. SmartLMS в папке Readings.Useful)
- **https://www.baeldung.com/java-method-handles**
- **https://www.baeldung.com/java-variable-handles**
- Для продвинутых-интересующихся:
  - Почитайте статью «**How JDK 10 var introduction have the anonymous inner classes usage**»): **https://medium.com/the-java-report/how-java-10-changes-the-way-we-use-anonymous-inner-classes-b3735cf45593** Там показано, что *type inference* для **var** может быть «***умнее автора программы***»... ☺
  - Полезный блог Владимира Долженко : **http://dolzhenko.blogspot.ru/**