Кориснички дефинисани изузеци

Постоје два основна разлога за дефинисање наших класа изузетака:

- желимо да додамо информације када се деси стандардни изузетак и можемо то учинити реизбацивањем објекта наше класе изузетака
- можемо имати грешке које настају у нашем кôду које оправдавају креирање специјалне класе изузетака.

Дефинисање наше класе изузетка

Наше класе изузетака морају увек имати `Throwable` за суперкласу. Ако као суперкласу користимо `RuntimeException`, компајлер неће проверавати да ли се наши изузеци хватају нити да ли методи декларишу да могу да доведу до њиховог избацивања.

Минимум који треба да обезбедимо у дефиницији наше класе изузетака су два конструктора:
подразумевани и конструктор који прима један аргумент типа `String`.
Атрибут типа `String`, који је базни део објекта (потиче од суперкласе `Throwable`) и представља поруку о грешци, биће аутоматски иницијализован именом наше класе без обзира на то који конструктор наше класе се користи.
Стринг прослеђен другом конструктору биће дописан на име класе.

Ово је минимум. Наравно, могу се додати и други конструктори, а генерално ћемо то и желети, посебно ако реизбацујемо наш изузетак након избацивања стандардног изузетка.

Типично ћемо желети да додамо и атрибуте који ће чувати додатне информације о проблему, као и методе који ће омогућити кôду `catch` блока да приступа тим подацима.

Пошто је наша класа обавезно изведена из класе `Throwable`, аутоматски ће јој бити доступне и информације са стека извршавања.

Избацивање нашег изузетка

Као и други изузеци, и наши се избацују наредбом `throw`.
Нпр.

```
public class GrozanProblemException extends Exception {
      // Konstruktori
      public GrozanProblemException(){}

      public GrozanProblemException(String s){
            super(s);  // poziv konstruktora bazne klase
      }
}
```

Након позива подразумеваног конструктора:

```
GrozanProblemException e = new GrozanProblemException();
throw e;
```

порука о грешци ће се састојати само од пуног квалификованог имена класе нашег изузетка, док,
уколико се позове други конструктор:

```
throw new GrozanProblemException("Strasne poteskoce!");
```

порука о грешци биће `String` у коме пише: `"GrozanProblemException: Strasne poteskoce!"`

Писање сопствених класа изузетака

Делови текста су прикупљени са више различитих места и дело су различитих аутора који немају сви потпуно исти став о свим детаљима коришћења сопствених типова изузетака. Не постоје чврста правила којих се треба држати и избор онога што ће бити коришћено у одређеном програму зависи од његовог контекста.

један од аутора

Треба да постоји уговорни однос између позивајућег и позваног метода.
Позивајући метод очекује да позвани метод изврши одређени задатак, а позвани метод очекује да му позивајући обезбеди податке које ће користити док извршава своје операције. Када било који од ових метода не испуни своје дужности, дешава се абнормална ситуација и треба избацити изузетак који се проверава (checked) или не проверава (unchecked exception).

Изузетак **који се проверава** означава да позвани метод није испунио свој део уговора. Пошто операције у позивајућем програму зависе од тога, избацивањем изузетка који се проверава, програмер позивајућег програма има могућност да рукује таквом аномалијом. Нпр. *IOException* је изузетак који се проверава, јер ако нешто није у реду приликом отварања фајла или писања у фајл, програмер мора да зна и да буде спреман да се носи са таквом катастрофом. Насупрот томе, *StringIndexOutOfBounds* је изузетак који се не проверава.
Изузеци који се не проверавају су типа поткласа од *RuntimeException*. Изузетак **који се не проверава** се избацује јер позивајући метод није испунио свој део уговора прослеђујући некоректне податке. Оставља се програмеру тог кôда да то открије и исправи, јер је проблем у његовој имплементацији, не у Вашој.
Да бисте имплементирали изузетке који се проверавају, морате навести све које позвани метод може избацити користећи кључну реч *throws*.

```
public static void example(Object input) throws MyCheckedException{
    /*
       throws indicates the names of checked exception classes
       that example() throws. It must be an Exception
       or a subclass that you or someone else has written thereof
    */

    if(input != valid){
      throw new MyRuntimeException("You didn't give me the info I wanted!");
      //throw unchecked exception
    }
    /*
      code performing example function
    */
    if(!performed)
      //performed is a boolean verifying successful operations
      throw new MyCheckedException("Oops, something serious happened on my end!");

    //Otherwise operation completed successfully
}
```

Када пишемо конструктор своје класе изузетака, можемо му проследити какву год хоћемо поруку о грешци и на тај начин креирати персонализоване поруке ради ефективне пријаве и отклањања грешака. Можемо чак проследити конструктору други *Throwable* објекат, попут *RuntimeException* или неки други *Exception* који је узроковао грешку на првом месту. Други методи (на вишем нивоу апстракције) могу искористити ове податке како би дали прецизнију слику узрока проблема.

```
public class MyCheckedException extends Exception{
    public MyCheckedException(String msg){
      super(msg);
    }

    public MyCheckedException(String msg, Throwable t){
      super(msg,t);
    }
}
```

Кадгод пишемо изведену класу, можемо додати шта год желимо у своју поткласу. То омогућује велику флексибилност. Нпр. методи за логовање грешака у фајл могу бити имплементирани на овом нивоу.

Дакле, могуће је користити произвољне структуре и поруке, а одговарајућим планирањем могуће је да описи грешака буду веома дескриптивни тако да крајњим корисницима дају релевантне информације казујући им шта није у реду, техничка подршка може моментално лоцирати и исправити проблеме, а програмери могу једноставније пронаћи и исправити сопствене грешке. Напреднији концепт оланчавања изузетака(exception chaining) заснива се на чињеници да је могуће енкапсулирати узрок изузетка тако да виши нивои могу доносити комплексније одлуке о руковању изузецима, а можда чак вршити и динамичку корекцију проблема. Могуће примене су безбројне.

Један од случајева када је уобичајена пракса избацити `RuntimeException` је када корисник позове метод неисправно. На пример, метод може проверити да ли је неки од његових аргумената `null`. Ако нађе такав аргумент, метод може избацити `NullPointerException`, а то је изузетак који се не проверава.

Савет је: ако клијент може да се опорави од изузетка, изузетак треба да буде од оних који се проверавају.
Ако клијент не може да предузме ништа да се опорави од изузетка, изузетак треба да буде од оних који се не проверавају.

''

Други аутор

Изузеци треба да буду непроменљиви објекти. У класи изузетака треба написати getter методе, а setter не, јер је њима могуће случајно или намерно променити објекат.

''

Трећи аутор

Јава користи изузетке како би пријавила озбиљне грешке. Под „озбиљним грешкама“ мисли се на ситуације у програму које програмер не сматра „нормалним“. То може бити изненадни прекид конекције са базом података, покушај дељења нулом или било шта друго што је витално за исправно функционисање програма. Оно што НИЈЕ озбиљна грешка је нпр. када добијемо крај фајла читајући фајл. Чак и ако се то не дешава често, очекује се да ће се десити и стога треба додати кôд у програм који ће руковати том ситуацијом.

Традиционално се у Јава литератури употреба изузетака илуструје случајем дељења нулом.
Пример служи само да би се илустровала техника. Било би једноставније, и коректније, пре дељења, тестирати да ли делилац има вредност 0.

Када ухватите изузетак имате неколико могућности:

- да поправите ситуацију тако да програм може да настави нормално да се извршава
- да избаците нови изузетак којим позивајући метод мора да рукује
- да игноришете грешку (не саветује се!)
- да позовете System.exit()--вероватно након што логујете информације о грешци

Могућност која је заиста корисна је могућност да креирате сопствене класе изузетака. Избацивањем својих изузетака упрошћавате кôд за руковање грешкама, пошто можете руковати системским и грешкама апликације на исти, конзистентан начин.

''

Изузеци типа Error и њених поткласа не могу се поправити у коду када се појаве. Могу се поправити променом окружења (попут ограничења меморије) или deployment-а (када класа недостаје или је погрешна њена верзија). Не хватати их.

''

Из http://www.javaspecialists.eu/archive/Issue162.html :

Савети како треба:

1. Не пишите сопствене изузетке. Можда је проблем једноставно могуће решити повратном вредношћу метода. Неки корисни стандардни изузеци који се могу користити уместо да се пишу своји су:

```
IllegalStateException
UnsupportedOperationException
IllegalArgumentException
NoSuchElementException
NullPointerException
```

   Тип изузетка који избацујете треба јасно да репрезентује околности због којих је избачен.

2. Пишите корисне изузетке. Постоје околности када треба прекршити први савет. Нпр. када желимо да имамо свеобухватнији изузетак који означава да се десио било који од неколико проблема. Када пишемо своје изузетке, треба да пратимо конвенцију да се имена класа завршавају са „Exception“. Чланице треба да буду final (то чини конкурентност једноставнијом). Наше класе изузетака треба да садрже више информација од обичног стринга. Нпр. ако желимо да дефинишемо изузетак који тврди да је вредност изван опсега, класа не треба да садржи само некоректну вредност, већ и опсег који се сматра коректним:

```
public class OutOfRangeException
    extends IllegalArgumentException {
  private final long value, min, max;

  public OutOfRangeException(long value, long min, long max) {
    super("Value " + value + " out of range " +
        "[" + min + ".." + max + "]");
    this.value = value;
    this.min = min;
    this.max = max;
  }

  public long getValue() {
    return value;
  }

  public long getMin() {
    return min;
  }

  public long getMax() {
    return max;
  }
}
```

   Ова класа се може користити како би се дефинисала класа Person, koja допушта године само између 0 и 150. Нпр.

```
public class Person {
  public static final int MIN_AGE = 0;
  public static final int MAX_AGE = 150;

  private final int age;
  private final String name;

  public Person(int age, String name) {
    this.age = age;
    this.name = name;
    if (this.age < MIN_AGE || this.age > MAX_AGE) {
      throw new OutOfRangeException(this.age, MIN_AGE, MAX_AGE);
    }
  }
```

```
  public String toString() {
    return "Person " + name + " is " + age + " years old";
  }
}
```

3. Избацујте изузетке рано. Изузетке је потребно избацити што је могуће пре. Чим се открије грешка, неопходно је генерисати изузетак.Ако се предуго чека, анализирање проблема постаје теже.

4. Хватајте изузетке касно. Изузетке треба хватати у контекстима где је могуће предузети неку корективну акцију. Иначе их треба прослеђивати навише (throws) све док не добијемо довољно информација да можемо да се носимо са њима.

5. Документујте изузетке. Методи и конструктори који могу избацити изузетке морају бити јасно документовани, укључујући савете кориснику који изузеци могу бити избачени под којим околностима. То олакшава клијентима да исправно поступају са изузецима. Изузетке који се не проверавају је, заправо, важније документовати, јер није могуће једноставно открити шта од њих може бити избачено гледајући само потпис метода. Нпр. за конструктор класе Person, из претходне дискусије, документација би била:

```
/**
 * Constructs a person with the given age and name.
 *
 * @param age  The age must fit into a range, specified by
 *             MIN_AGE and MAX_AGE
 * @param name The name should not be null, nor an empty
 *             string
 * @throws OutOfRangeException      if the age is out of range.
 *                                  The age may not be less
 *                                  than the constant MIN_AGE
 *                                  and may not be more than
 *                                  the constant MAX_AGE.
 * @throws IllegalArgumentException if the name is null or
 *                                  empty.
 * @see #MIN_AGE, #MAX_AGE
 */
public Person(int age, String name) {
  this.age = age;
  this.name = name;
  if (this.age < MIN_AGE || this.age > MAX_AGE) {
    throw new OutOfRangeException(this.age, MIN_AGE, MAX_AGE);
  }
  if (this.name == null || this.name.equals("")) {
    throw new IllegalArgumentException(
      "name parameter should not be null nor empty");
  }
}
```

Савет како НЕ ТРЕБА:

Никада не треба изазивати изузетак чије избацивање је могуће спречити. Аутор је виђао кôд у коме се, уместо провере граница, претпоставља да ће подаци бити коректни и хватају RuntimeException изузеци. Пример лошег кода (НЕ ПРОГРАМИРАЈТЕ ОВАКО!)

```
public class Antipattern1 {
  public static void main(String[] args) {
    try {
      int i = 0;
      while (true) {
        System.out.println(args[i++]);
      }
    } catch (ArrayIndexOutOfBoundsException e) {
      // we are done
    }
  }
}
```