МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧЕРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет информационных технологий и программирования

**Лабораторная работа № 6**

Выполнил студент группы № M3307  
Бойцов Виталий

САНКТ-ПЕТЕРБУРГ  
2019

# Упражнение 6-1. Обработка исключительных ситуаций.

Цель упражнения: Изучить способы обработки исключительных ситуаций в программах на Java.

Описание упражнения: В этом упражнении вы объявите в проекте несколько собственных типов исключений и используете их в методах классов проекта.

1. Объявите в пакете ru.itmo.exceptions следующие классы исключений:
   1. CatalogLoadException : исключение, выбрасываемое методом load интерфейса CatalogLoader. Выбрасывается при любых ошибках загрузки каталога.
   2. ItemAlreadyExistsException : исключение, выбрасываемое методом addItem класса ItemCatalog. Выбрасывается в том случае, когда добавляемая позиция уже присутствует в каталоге.
   3. NegativeQuantityException : исключение, выбрасываемое методом remQuantity класса Warehouse (будет создан в последующих упражнениях). Выбрасывается в случае попытки списать со склада количество товара больше имеющегося.   
      Примечание: Все эти классы являются проверяемыми исключениями и должны наследовать классу Exception.
2. Предусмотрите в методе load интерфейса CatalogLoader и реализующего его класса CatalogStubLoader выброс исключения CatalogLoadException. Для класса CatalogStubLoader код должен предусматривать ошибки на этапе добавления позиции в каталог:   
   try {   
     cat.addItem(item1);   
     cat.addItem(item2); }   
   catch (ItemAlreadyExistsException e) { // TODO Auto-generated catch block   
     e.printStackTrace();    
     throw new CatalogLoadException(e);   
   }
3. Предусмотрите в методе addItem класса ItemCatalog выброс исключения ItemAlreadyExistsException. Исключение должно выбрасываться в том случае, когда добавляемая позиция уже есть в каталоге.
4. Исправьте все ошибки, появившиеся в проекте в связи с необходимостью предусмотреть обработку новых исключений.
5. Запустите проект и проверьте его работоспособность. Протестируйте ситуации, в которых могут быть выброшены, добавленные в проект исключения.
6. package ru.billing.exceptions;  
     
   public class CatalogLoadException extends Exception{  
    public CatalogLoadException(Exception e) {  
    super(e);  
    }  
   }

package ru.billing.exceptions;  
  
public class ItemAlreadyExistsException extends Exception{  
}

package ru.billing.exceptions;  
  
public class NegativeQuantityException extends Exception{  
}

# Упражнение 6-2 Синхронизация потоков.

Цель упражнения: Изучить способы синхронизации работы нескольких потоков в многопоточной среде JVM.

Описание упражнения: В этом упражнении вы смоделируете работу банковских операций со счетами.

1. Добавьте в проект новый пакет sync, в котором буде создавать все классы из этого упражнения
2. Создайте класс U1901Bank, в котором будем имитировать работу банковской операции со счетами и наполните его следующим кодом:
   1. Объявите две int-переменные уровня экземпляра класса с именами intTo и intFrom, которые будут имитировать счет-отправитель и счет-получатель (кредит и дебет). Переменную intFrom инициализируйте значением 220.
   2. Создайте метод без возвращаемого значения с именем calc. В этом методе будет организована банковская операция по переброске денег между счетами (между переменными intFrom и intTo). Между снятием денег с одного счета (уменьшением значения переменной intFrom) и пополнением другого счета (увеличением переменной intTo) будет организована задержка. длительность которой передается в качестве входного параметра. Также в качестве входного параметра передается сумма перевода между счетами. Наполните метод следующим кодом:
      1. Укажите два входных параметра – intTransaction (типа int) для передачи суммы и lngTimeout (типа long) для указания длительности временной задержки.
      2. Для контроля работы метода выведите сообщение, в котором укажите начальные значения переменных intTo и intFrom, а также имя текущего потока (при помощи метода Thread.currentThread().getName().
      3. Уменьшите значение переменной intFrom на значение переменной intTransaction и сохраните результат в той же переменной intFrom
      4. Организуйте временную задержку на lngTimeout миллисекунд при помощи метода Sleep класса Thread. Оберните вызов этого метода в try/catch.
      5. Увеличьте значение переменной intTo на значение переменной intTransaction и сохраните результат в той же переменной intTo
      6. В завершении выведите имя текущего потока и изменившиеся значения переменных (подобно пункту 2.ii). Рекомендуется в обоих случаях указать какое-нибудь уникальное слово (например before/after) для идентификации вывода.
3. Создайте класс-наследник от Thread с именем U1901Thread, в котором будем реализовывать многопоточность для работы с методом calc класса U1901Bank. Наполните класс следующим кодом:
   1. Объявите следующие переменные уровня экземпляра класса:
      1. bankWork типа U1901Bank
      2. intTrans типа int
      3. lngSleep типа long.
   2. Создайте конструктор, в котором должно быть три параметра для заполнения этих переменных уровня экземпляра класса. Тип параметров должен соответствовать типу переменной, имена параметров можете указать сами.  
      Такой извилистый способ требуется потому, что метод run(), который используется для многопоточной работы, не имеет параметров.
   3. Создайте метод run(), в коде которого вызовите метод calc объекта bankWork и передайте в этот метод значения переменных intTrans и lngSleep.
4. Создайте класс U1901Main, который будет вызываться из-под JVM для проверки работоспособности многопоточности. В этом классе будет один метод, имя его предлагается определить самостоятельно. В этом методе делаем следующее:
   1. Создать экземпляр класса U1901Bank с именем bankMain и инициализировать его конструктором.
   2. Создайте экземпляр класса U1901Thread с именем threadOne и инициализируйте его конструктором со следующими параметрами: bankMain, 100,2000.
   3. Задайте этому экземпляру уникальное и понятное имя потока при помощи метода setName и приоритет Thread.MAX\_PRIORITY при помощи метода setPriority.
   4. Запустите поток при помощи метода start()
   5. Повторите действие пунктов b,c,d для экземпляра того же класса, но с именем threadTwo, другим именем потока, конструктором с параметрами: bankMain, 50,300.
   6. В качестве контрольного выстрела, в конце метода можно вывести имя текущего потока – Thread.currentThread().getName(). Имена всех трех потоков должны быть разными!
5. Запустите класс U1901Main. Вывод должен быть примерно таким -   
   before Thread=thread\_100, From=220, To=0  
   before Thread=thread\_50, From=220, To=0  
   End, main  
   After Thread=thread\_50, From=70, To=50  
   After Thread=thread\_100, From=70, To=150  
   Что здесь не правильно – в выделенной строке сумма значений переменных отличается от суммы в других строках. Это плохо – деньги где-то зависли. Хозяева могут дать по шее.
6. Для исправления сего недостатка надо сделать метод calc синхронизированным. После этого вывод будет таким:  
   End, main  
   before Thread=thread\_100, From=220, To=0  
   After Thread=thread\_100, From=120, To=100  
   before Thread=thread\_50, From=120, To=100  
   After Thread=thread\_50, From=70, To=150
7. Ура, мы победили – у нас везде 220!
8. package sync;  
     
   public class U1901Bank {  
    private static int *intTo*;  
    private static int *intFrom* = 220;  
     
    public U1901Bank() {  
    }  
     
    public synchronized void calc(int intTransaction, long lngTimeout, int i) {  
    System.*out*.println("Before:\n" + "intTo: " + *intTo* + " intFrom: " + *intFrom* + " CurrentThread: " + Thread.*currentThread*().getName() + " instance " + i);  
    *intFrom* -= intTransaction;  
    try {  
    Thread.*sleep*(lngTimeout);  
    } catch (InterruptedException e) {  
    e.printStackTrace();  
    }  
    *intTo* += intTransaction;  
    System.*out*.println("After:\n" + "intTo: " + *intTo* + " intFrom: " + *intFrom* + " CurrentThread: " + Thread.*currentThread*().getName() + " instance " + i);  
    }  
     
    public void not\_sync(int intTransaction, long lngTimeout) {  
    System.*out*.println("In not sync before:\n" + "intTo: " + *intTo* + " intFrom: " + *intFrom* + " CurrentThread: " + Thread.*currentThread*().getName());  
    *intFrom* -= intTransaction;  
    try {  
    Thread.*sleep*(lngTimeout);  
    } catch (InterruptedException e) {  
    e.printStackTrace();  
    }  
    *intTo* += intTransaction;  
    System.*out*.println("In not sync after:\n" + "intTo: " + *intTo* + " intFrom: " + *intFrom* + " CurrentThread: " + Thread.*currentThread*().getName());  
    }  
   }

package sync;  
  
public class U1901Thread extends Thread {  
 U1901Bank bankWork;  
 U1901Bank bankWork2;  
 int intTrans;  
 long lngSleep;  
  
 public U1901Thread(U1901Bank bank, int trans, long sleep) {  
 bankWork = bank;  
 intTrans = trans;  
 lngSleep = sleep;  
 bankWork2 = new U1901Bank();  
 }  
  
 @Override  
 public void run() {  
 System.*out*.println("execute");  
 synchronized (U1901Bank.class) {  
 bankWork.calc(intTrans, lngSleep, 1);  
 bankWork2.calc(intTrans, lngSleep, 2);  
 }  
  
 }  
}

package sync;  
  
public class U1901Main {  
 public static void main(String[] args) {  
 U1901Bank bankMain = new U1901Bank();  
 U1901Thread threadOne = new U1901Thread(bankMain, 100, 2000);  
 threadOne.setName("First thread");  
 threadOne.setPriority(Thread.*MAX\_PRIORITY*);  
 threadOne.start();  
  
 U1901Thread threadTwo = new U1901Thread(bankMain, 50, 300);  
 threadTwo.setName("Second thread");  
 threadTwo.setPriority(Thread.*MAX\_PRIORITY*);  
 threadTwo.start();  
  
 System.*out*.println("End of " + Thread.*currentThread*().getName());  
 }  
}

# Упражнение 6-3-1 Потоки ввода -вывода.

Цель упражнения: Научиться работать с потоками ввода-вывода.

Описание упражнения: В этом упражнении вы реализуете загрузчик каталога товаров из структурированного текстового файла.

1. Добавьте в пакет ru.itmo.client новый класс CatalogFileLoader, реализующий интерфейс CatalogLoader.
2. Добавьте в класс CatalogFileLoader строковое поле, содержащее имя файла со списком товаров и конструктор, принимающий на вход имя этого файла:  
   private String fileName;  
   public CatalogFileLoader(String fileName) {  
     this.fileName = fileName;  
   }
3. Реализуйте метод load класса CatalogFileLoader следующим образом:  
   File f = new File(fileName);  
   FileInputStream fis;  
   String line;  
    try {  
     fis = new FileInputStream(f);  
     Scanner s = new Scanner(fis);  
     
     while(s.hasNextLine()){  
     line = s.nextLine();  
     if(line.length()==0) break;  
     String[] item\_fld = line.split(";");  
     String name = item\_fld[0];  
     float price = Float.parseFloat(item\_fld[1]);  
     short expires = Short.parseShort(item\_fld[2]);  
     FoodItem item = new FoodItem(name,price,null, new Date(),expires);  
     cat.addItem(item);  
     }  
    } catch (FileNotFoundException e) {  
     e.printStackTrace();  
     throw new CatalogLoadException(e);  
    } catch (ItemAlreadyExistsException e) {  
     e.printStackTrace();  
     throw new CatalogLoadException(e);  
    }  
   }
4. package ru.billing.client;  
     
   import ru.billing.exceptions.CatalogLoadException;  
   import ru.billing.exceptions.ItemAlreadyExistsException;  
   import ru.billing.stocklist.FoodItem;  
   import ru.billing.stocklist.ItemCatalog;  
     
   import java.io.File;  
   import java.io.FileInputStream;  
   import java.io.FileNotFoundException;  
   import java.util.Date;  
   import java.util.Scanner;  
     
   public class CatalogFileLoader implements CatalogLoader {  
    private String fileName;  
     
    public CatalogFileLoader(String fileName) {  
    this.fileName = fileName;  
    }  
     
    @Override  
    public void load(ItemCatalog cat) throws CatalogLoadException {  
    File f = new File(fileName);  
    FileInputStream fis;  
    String line;  
    try {  
    fis = new FileInputStream(f);  
    Scanner s = new Scanner(fis);  
    while (s.hasNextLine()) {  
    line = s.nextLine();  
    if (line.length() == 0) break;  
    String[] item\_fld = line.split(";");  
    String name = item\_fld[0];  
    float price = Float.*parseFloat*(item\_fld[1]);  
    short expires = Short.*parseShort*(item\_fld[2]);  
    FoodItem item = new FoodItem(name, price, null, new Date(), expires);  
    cat.addItem(item);  
    }  
    } catch (FileNotFoundException e) {  
    e.printStackTrace();  
    throw new CatalogLoadException(e);  
    } catch (ItemAlreadyExistsException e) {  
    e.printStackTrace();  
    throw new CatalogLoadException(e);  
    }  
    }  
   }