18.04 Vid\_1\_2 Exceptions & Errors Handling in Java /Обработка ошибок и исключительных ситуаций

1.General concept

2.Using try/catch and finally blocks

3.Using throw & throws

4.Checked & Unchecked Exceptions

5.Create Own Exceptions

6.Assertions

7.Whats new in JDK 7.0

Главный посыл обработки исключений, в том, что оно может произойти, нужно предусмотреть реакцию на него: обработка или делегирование(проброс)(caught or declare to be thrown).

Exception Handling

Method where error occurred

Method where error occurred

Method call Looking for appropriate

Method whitout an exception handling

Throws exception handler

Method whitout an exception handling

Method call

Method whitout an exception handling

Forward exception Looking for appropriate

Method whitout an exception handling

Method call handler

main()

Catches some other exception

main()

Если в системе отсутствует обработчик exception,то он будет обрабатываться дефолтным обработчиком JVM, который аварийно завершает выполнение и выводит StackTrace(путь возникновения исключения)

Java Exceptions Hierarchy

Объекты которые могут инициировать запуск обработки исключений, только Throwable !!!

**Throwable**

**Error**

System problems (JVM) Application problems

**Exception**

Не обрабатываются, информ. Функция

Инициатором механизма обработки исключений может быть как JVM , так и пользователь с помощью оператора throw, который прерывает выполнение программы и переходит к обработке.

Для реализации механизма обработки исключений исполльзуется 5 ключевых слов:

Exception Handling: try, catch, finally, throw, throws

Основная структура обработки:

Потенциально опасный блок кода, помещается в блок try, в {};

Catch – блок обработки,в условии декларируется тип исключения в (), а в {}описание ветки

try{ //code in which exceptions may be thrown } catch (Exception e){ }//code executed if exception occurs }

public class Test {

public static void main(String args[]) {

int[] array = { 1, 2, 3 };

try {

array[10] = 10; ArrayIndexOutOfBoundsException

System.out.println("this won't be displayed..."); 🡪 этот код выполнятся не будет

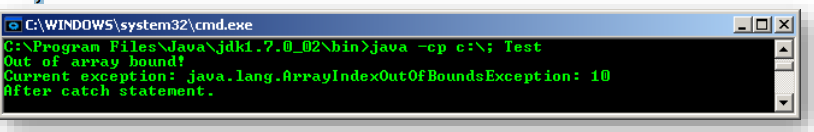
}catch (ArrayIndexOutOfBoundsException e)

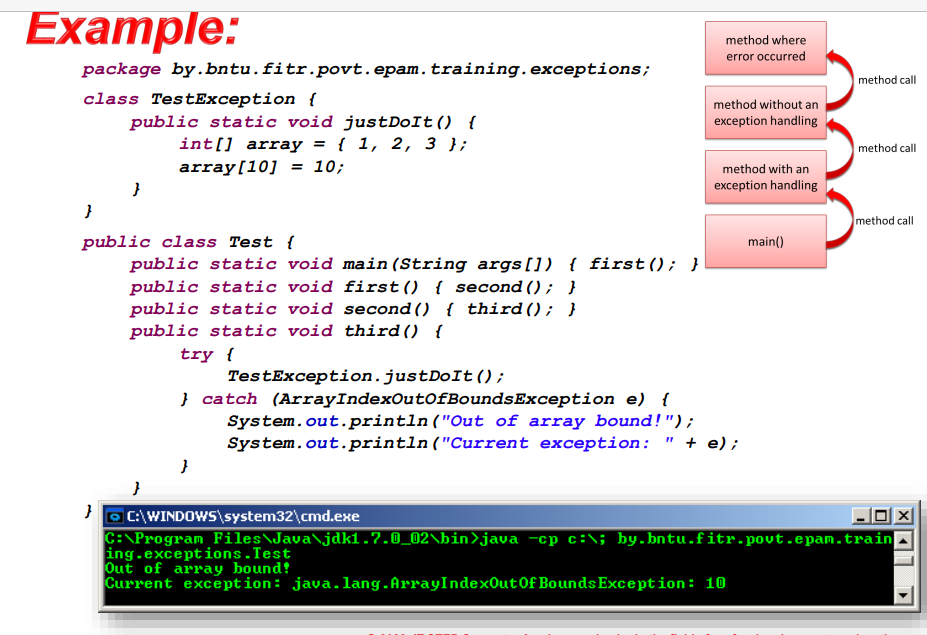
{ System.out.println ("Out of array bound!"); сразу переходим к блоку catch

System.out.println("Current exception: " + e);

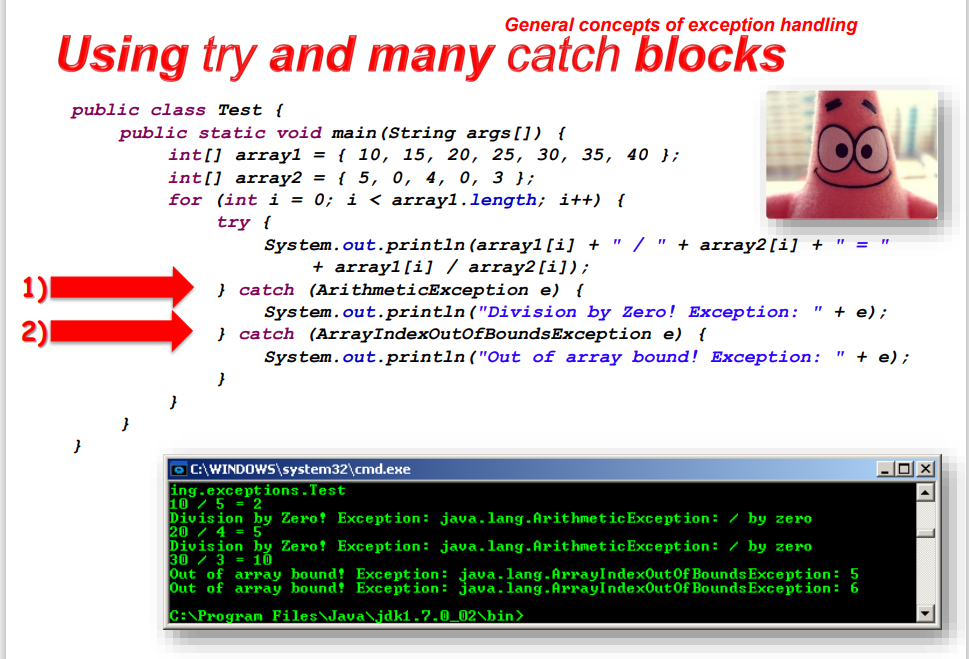
}

System.out.println("After catch statement."); } переходим к дальнейшему выполнению

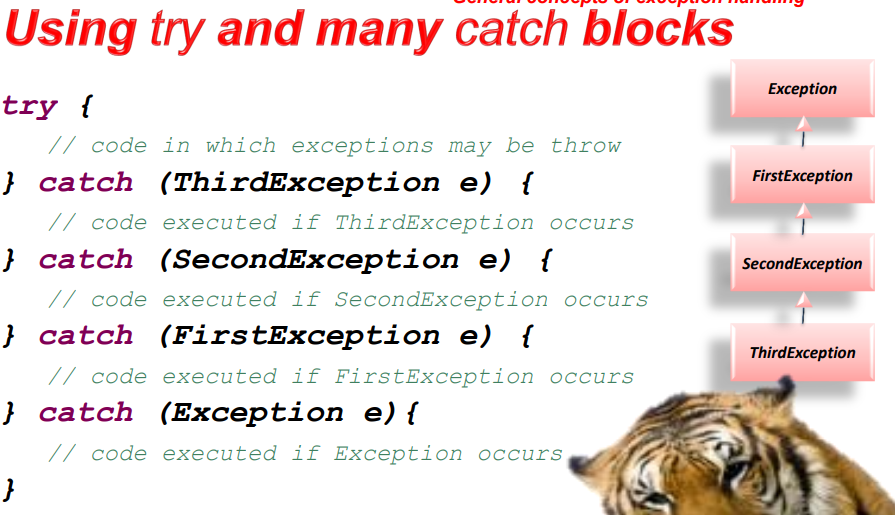




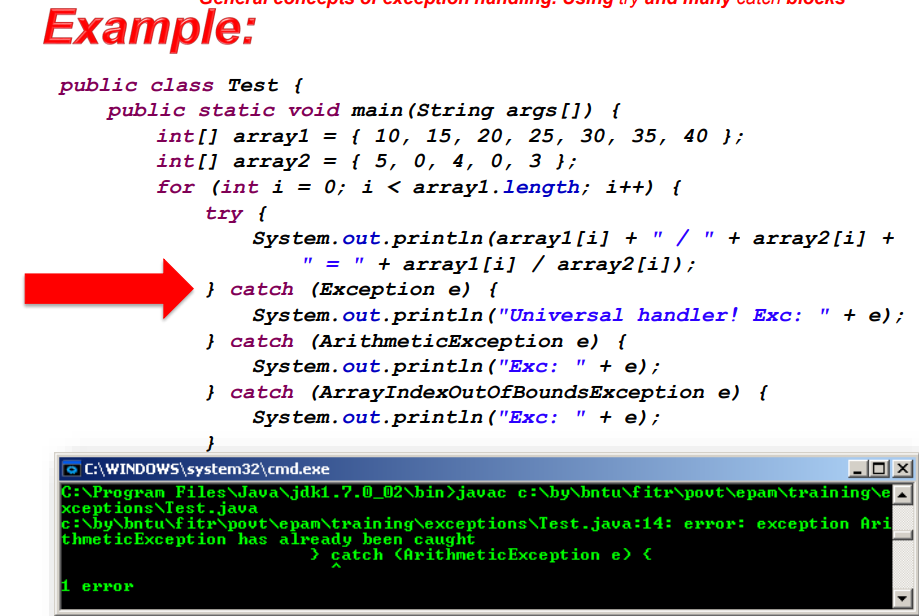
Обработка нескольких исключений происходит путем размещения их в разных блоках catch, относящихся к одному блоку try



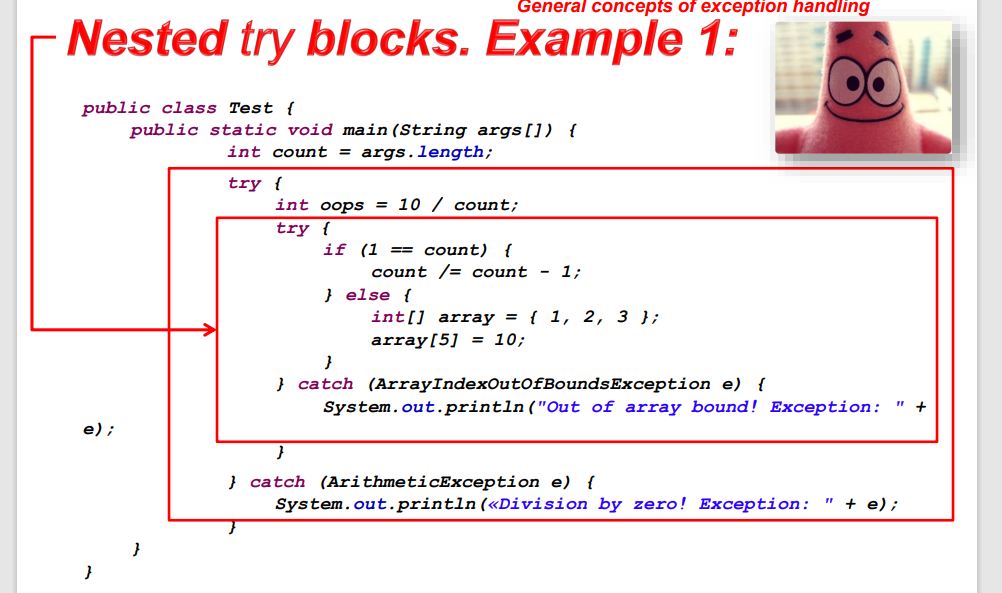
В случае отсутствия обработки данных исключений, программа прекратила бы свою работу, а при наличии обработчика, программа продолжает работать с корректными данными

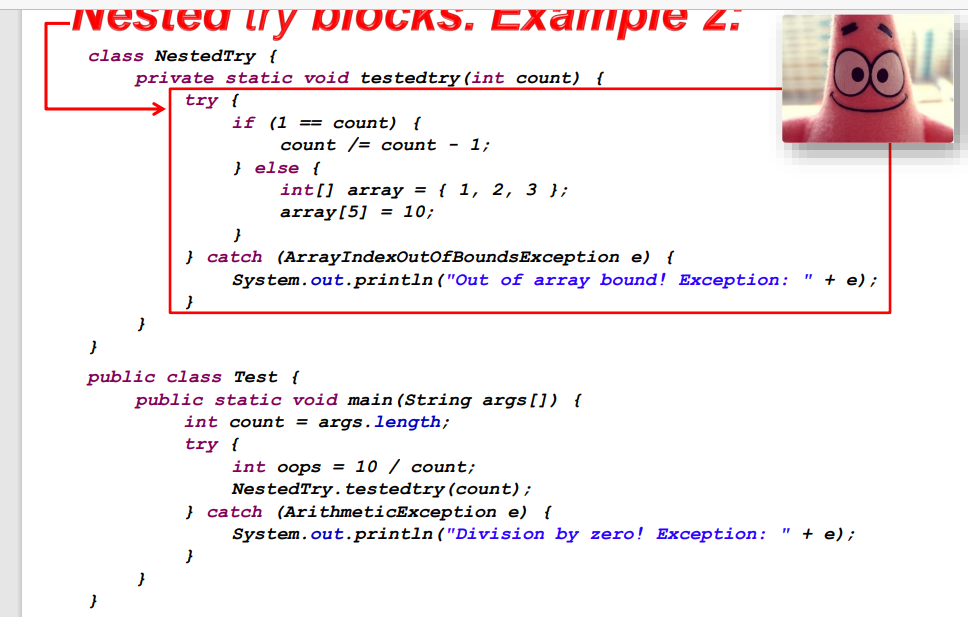
При обработке исключений – родственников, их обработку следует размещать в порядке обратном наследованию, 

тк иначе базовый класс будет перехватывать исключения, относящиеся к наследникам (ссылке базового класса можно присвоить объект производного), а наследники не смогут получить управление и перейдут в разряд недостижимого кода, что приведет к ошибке компиляции



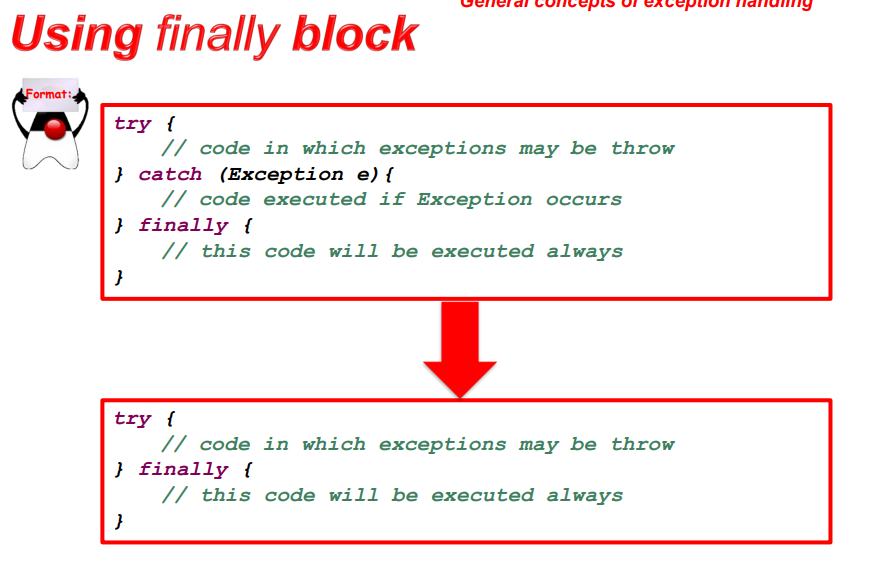
Использовать вложенные блоки try не рекомендуется, но все же иногда для ускорения тестирования общего функционала программы(для заказчика), практикуется:

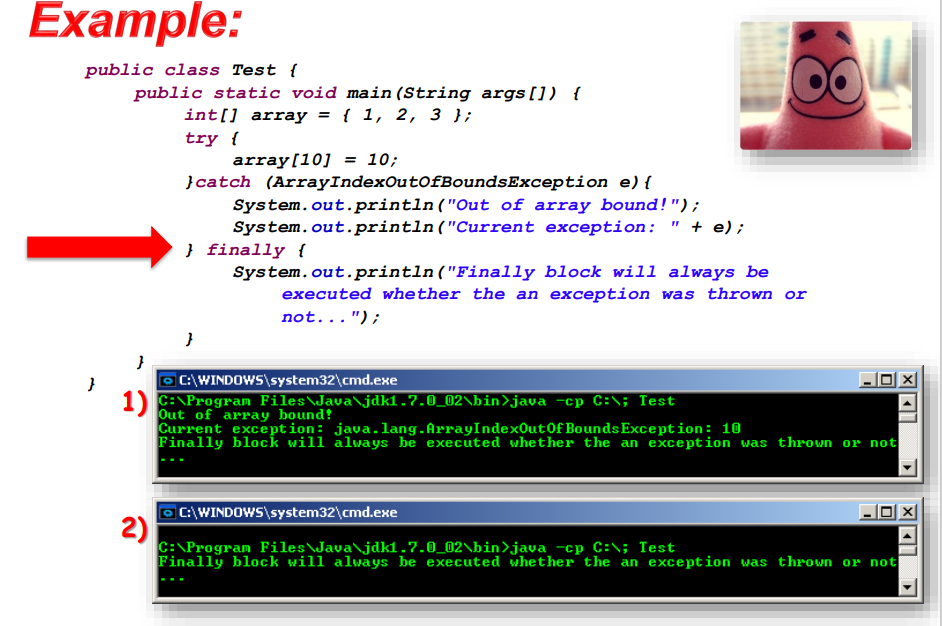




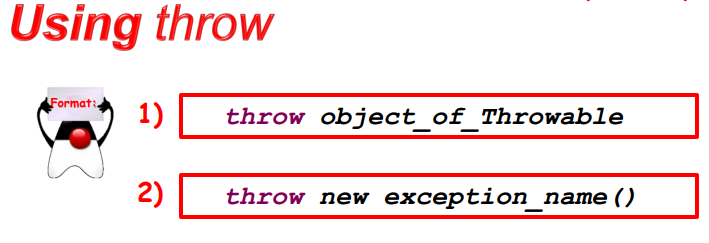
Лучше конечно разделять, для облегчения обслуживания

Использование finally, завершающий процесс обработки(выполняется всегда), может использоваться как с обработчиком так и самостоятельно(например для освобождения внешнего ресурса(БД), не подвластного JVM, следовательно и Garbage Collector)

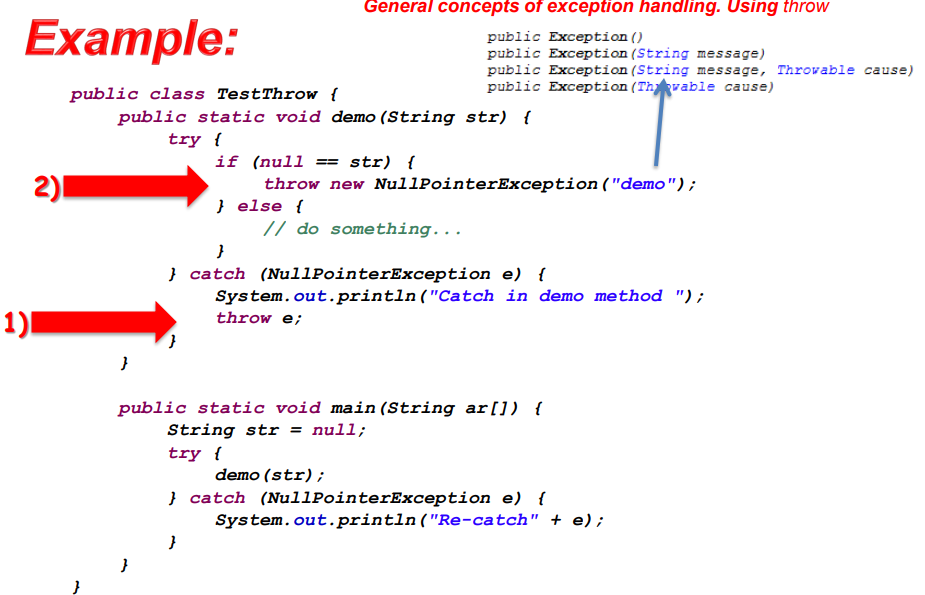




Пробрасывание исключений происходит,путем декларации с помощью throws, 2 способа:

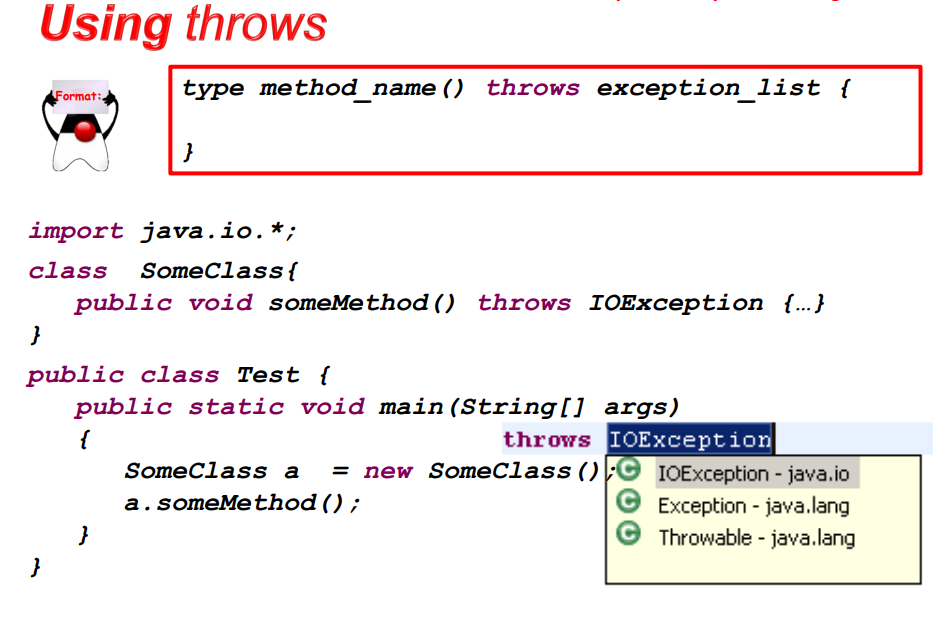


Rethrow, повторно создается объект

Налету создается анонимный объект

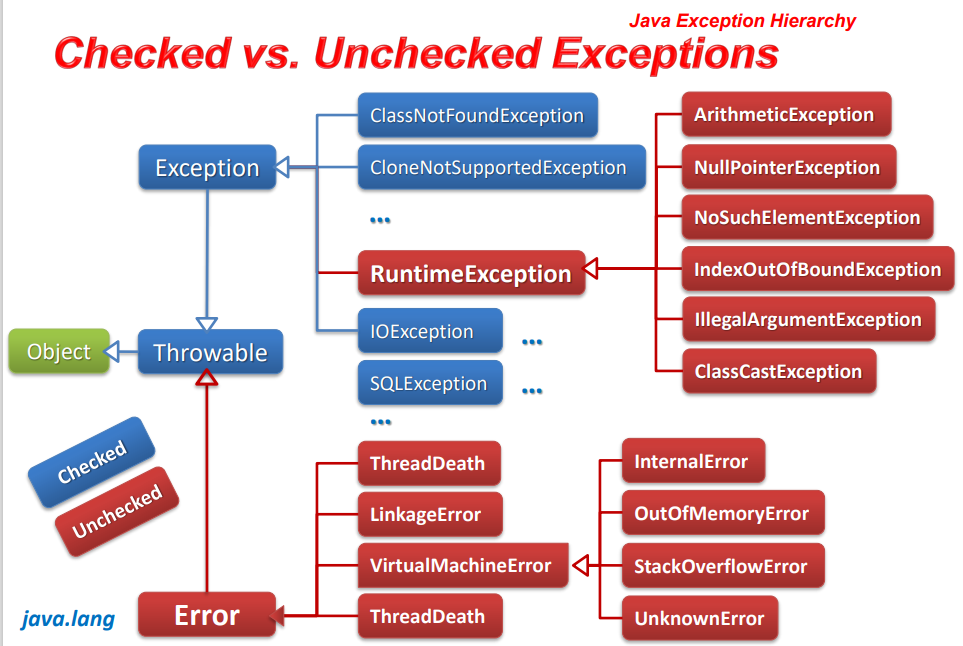
На практике не используются стандартные исключения, создавать свои!!!

Механизм делегирования: К объявлению метода добавляется кс throws и список возможных исключений, которые могут произойти в методе, но не отлавливаются им…, что обязует пользователя метода, обработать или пробросить исключение дальше



Vid\_2

Checked & Unchecked Exceptions

Исключения, бывают критически важные и те от которых можно избавится workflow code (условной конструкцией)

Иерархия исключений

Error все uncheck, тк на уровне программы, сделать ничего с ними нельзя, тж не требуют обработки RuntimeException, причины возникновения RuntimeExc рекомендуется обрабатывать условной конструкцией

package itStep.yandr.javaLessons.lesson30.controller;  
  
import java.util.Scanner;  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 int[] array = {3, 6, 5, 8, 3,};  
 System.*out*.println("input: ");  
 int index = scanner.nextInt();  
  
 System.*out*.println("your item: " + array[index]);  
 }  
}

конструкцию можно защитить обработкой исключения

package itStep.yandr.javaLessons.lesson30.controller;  
  
import java.util.Scanner;  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 int[] array = {3, 6, 5, 8, 3,};  
 while (true) {  
 try {  
 System.*out*.println("input: ");  
 int index = scanner.nextInt();  
 System.*out*.println("your item: " + array[index]);

break;  
 } catch (ArrayIndexOutOfBoundsException e) {  
 System.*out*.println("Error.Try again...");  
 }catch (RuntimeException e){  
 System.*out*.println("Wow!!!RuntimeException.Try again...");  
 }  
 }  
 }  
}

Но это ресурсо-затратно, по сравнению с превентивной защитой условной конструкции

public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 int[] array = {3, 6, 5, 8, 3,};  
 while (true) {  
 System.*out*.print("\ninput: ");  
 int index = scanner.nextInt();  
 if (index <= 0 && index < array.length) {  
 System.*out*.println("your item: " + array[index]);  
 break;  
 } else {  
 System.*out*.println("Error.Try again...");  
 }  
 }  
 }

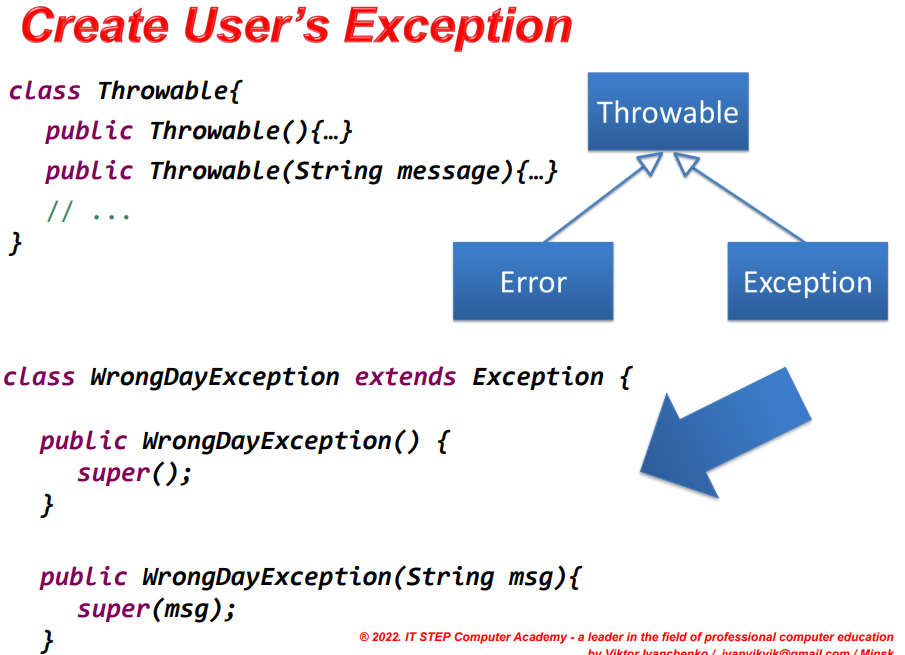
Предотвращение ClassCastException:

Main m;  
m.getClass().getName();

public class Main {  
 private int number;  
 private boolean flag;  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Main main = (Main) o;  
 return number == main.number && flag == main.flag;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(number, flag);  
 }

Создание собственных исключений:

Обязательно наследуются от Exception, для обязательной обработки, реализация как правило проста

на

Создание иерархии исключений:

Если относится к логике, то в модели( базовое), если наследники относятся к сущности, то в ней

LogicalStudentManagmenException

StudentManagmentProjectException

manual

\*\*TechnicalStudentManagmentException

StudentAgeWrongException\*

\*Логические –ручная генерация, зависят не от стандартного поведения JVM, а только от Вашей системы, например возраст не может быть отрицательным числом, реализуются с 2мя конструкторами

If (age<0||age>130){

Throw new StudentAgeWrongException}

\*\*Технические – генерируется на базе первоисточников от системы, источником будет исключение самой платформы, обрамленные в собственное исключение, с конкретизацией места возникновения реализуются с 4мя конструкторами

while (true) {  
 try {  
 System.*out*.println("input: ");  
 int index = scanner.nextInt();  
 System.*out*.println("your item: " + array[index]);  
 break;  
 } catch ( IOException e) {  
 System.*out*.println("Error.Try again...");  
 throw new StudentDataWrongReadException(e);  
 }catch (RuntimeException e){  
 System.*out*.println("Wow!!!RuntimeException.Try again...");  
 }  
 }  
}

Логически:

В памяти:

Пример:

