ЛАБОРАТОРНАЯ РАБОТА №9 РАБОТА С ИСКЛЮЧЕНИЯМИ.

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:

Целью данной лабораторной работы являются получение практических навыков разработки программ, изучение синтаксиса языка Java, освоение основных конструкций языка Java (циклы, условия, создание переменных и массивов, создание методов, вызов методов), а также научиться осуществлять стандартный ввод/вывод данных.

Ключевые слова: try, catch, finally, throw, throws

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:

Механизм исключительных ситуаций в Java поддерживается пятью ключевыми словами:

- **try**
- **catch**
- **finally**
- **throw**
- **throws**

В Java всего около 50 ключевых слов, и пять из них *связано* с *исключениями*: try, catch, finally, throw, throws. Из них catch, throw и throws применяются к экземплярам класса, причём работают они только с Throwable и его наследниками.

На рисунке 1 представлена иерархия классов исключений, используемая в Java.

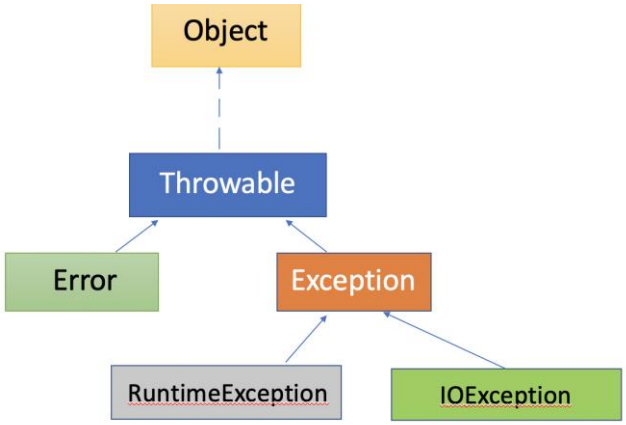

Рисунок 1 иерархия классов исключений

Наиболее популярные исключений в Java представлены в таблице 1.

Таблица 1. Классы исключений в Java

№ пп	Класс исключения	Класс предок/тип
1.	ArithmeticException	RuntimeException
2.	NegativeArraySizeException	RuntimeException
3.	ArrayIndexOutOfBoundsException	RuntimeException
4.	NoSuchElementException	RuntimeException
5.	ArrayStoreException	RuntimeException
6.	NotSerializableException	Exception
7.	AssertionError	Error
8.	NullPointerException	RuntimeException
9.	ClassCastException	RuntimeException
10.	NumberFormatException	RuntimeException
11.	ClassNotFoundException	Exception
12.	OutOfMemoryError	Error
13.	CloneNotSupportedException	Exception
14.	SecurityException	RuntimeException
15.	ConcurrentModificationException	RuntimeException
16.	StackOverflowError	Error
17.	EOFException	Exception
18.	StringIndexOutOfBoundsException	RuntimeException

19.	FileNotFoundException	Exception
20.	ThreadDeath	Error
21.	IllegalArgumentException	RuntimeException
22.	UnsupportedEncodingException	Exception
23.	InterruptedException	Exception
24.	UnsupportedOperationException	RuntimeException

То, что исключения являются объектами важно по двум причинам:

1) они образуют иерархию с корнем java.lang.Throwable (java.lang.Object — предок java.lang.Throwable, но Object —это не исключение!)
2) они могут иметь поля и методы

По первому пункту: catch — полиморфная конструкция, т.е. catch по типу класса родителя перехватывает исключения для экземпляров объектов как родительского класса, так или его наследников(т.е. экземпляры непосредственно самого родительского класса или любого его потомка).

Пример:

```
public class App {

   public static void main(String[] args) {

      try {

         System.err.print(" 0");

         if (true) {throw new RuntimeException();}

         System.err.print(" 1");

      } catch (Exception e) {  // catch по Exception ПЕРЕХВАТЫВАЕТ RuntimeException

           System.err.print(" 2");

        }

      System.err.println(" 3");

    }// end main

}
```

Результат работы программы:

>> 0 2 3

Упражнения (Основы Try-Catch- Finally)

Задание № 1

Выполните следующую программу и посмотрите, что происходит:

```
  public class Exception1 {

      public void exceptionDemo() {

        System.out.println( 2 / 0 );

      }

  }
```

Листинг 1

Необходимо инстанцировать класс и выполнить exceptionDemo (). Программа даст сбой, и вы получите следующее сообщение

java.lang.ArithmeticException: / by zero at Exception1.exceptionDemo(Exception1.java:12)

Это говорит нам о том, что программа пытается выполнить деление на ноль, который он не в состоянии выполнить.

Интересный эксперимент

Замените 2/0 на 2,0 / 0,0 и повторно вызовите метод. Что произойдет?

Теперь измените код в Exception1 чтобы включить try-catch block следующим образом:

```
public class Exception1 {

   public void exceptionDemo() {

     try{
          System.out.println( 2 / 0 );

        } catch ( ArithmeticException e ) {
           System.out.println("Attempted division by zero");

        }

   }

}
```

Листинг 2
Запустите программу и обратите внимание на новое поведение.

Задание № 2
Измените код в листинге 1 на следующий:

```
public class Exception2 {

    public void exceptionDemo() {

        Scanner myScanner = new Scanner( System.in );

        System.out.print( "Enter an integer ");
        String intString = myScanner.next();
        int i = Integer.parseInt(intString); System.out.println( 2 / i );

     }

}
```

Листинг 3
Запустите эту программу со следующими выводами: Qwerty 0 1.2 1
Посмотрите на вывод. Какие исключения выбрасываются?
Измените код, добавив блоки try – catch, чтобы иметь дело с определяемыми исключениями**.**

Задание № 3
С помощью перехватывания исключений можно оказывать влияние на поведение программы. В вашем решении в предыдущем упражнении вы можете добавить новый пункт - catch в начале списка пунктов catch. Выполите это, чтобы поймать общее исключение класса Exception. Перезапустите программу с приведенными выше данными и обратите внимание на ее поведение. Объясните новое поведение программы

Задание № 4
И наконец добавьте блок finally к решению Задания №2. Повторно запустите программу, чтобы проверить ее поведение. Объясните новое поведение программы

Генерация собственных исключений

На предыдущем шаге при выполнении заданий мы рассмотрели, как Java работает с предопределенными исключениями, теперь перейдем к тому, как генерируется исключение.

Все исключения в рассмотренных ранее примерах и заданиях были определены заранее. В этом разделе лабораторной работы вы будете создавать и пробрасывать свои собственные исключения (exceptions).

Задание № 5
Самый простой способ генерации исключения показан в следующем примере кода:

```
public class ThrowsDemo {

    public void getDetails(String key) {

        if(key == null) {
          throw new NullPointerException( "null key in getDetails" );

         }

       // do something with the key

      }

}
```

Листинг 4

Когда определяется условие ошибки, то мы выбрасываем исключение с определенным именем. Сообщение может быть связано с исключением. Откомпилируйте этот класс, создайте его экземпляр и выполните метод getDetails() с нулем в качестве значения параметра.

Вывод может быть следующим:

java.lang.NullPointerException: null key in getDetails at ThrowsDemo.getDetails(ThrowsDemo.java:13)

Добавьте блок try-catch таким образом, чтобы перехватить исключение и рассмотреть его внутри метода.

Подумайте, является ли этот способ подходящим, чтобы иметь дело с этим исключением? Объясните поведение программы.

Причиной ошибки, может является, например неправильное значение для параметра. Может было бы лучше, если бы метод вызывал getDetails() и там решалась бы эта проблема.

Обратите внимание на следующее:

```
public class ThrowsDemo {

    public void printMessage(String key) {

      String message = getDetails(key); System.out.println( message );

    }
    public String getDetails(String key) {
        if(key == null) {

            throw new NullPointerException( “null key in getDetails” );

         }

    return “data for” + key; }

}
```

Листинг 5

Откомпилируйте и запустите эту программу с правильным значением для ключа и с нулем в качестве значения. При выполнении с нулевым значением вы увидите некоторый вывод. Обобщите все вышесказанное и выполните эту программу с правильным значением для ключа и с нулем в ключе.

java.lang.NullPointerException: null key in getDetails
at ThrowsDemo.getDetails(ThrowsDemo.java:21)
at ThrowsDemo.printMessage(ThrowsDemo.java:13)

Теперь добавьте блоки try-catch, чтобы использовать для вывода сообщений метод printMessage(), таким образом, чтобы исключения обрабатывались и программа не “ломалась”.

Задание № 6

Теперь мы расширим наш пример для демонстрации прохождения исключения через цепочку вызовов. Создайте следующий класс и попытатайтесь его скомпилировать:

```
public class ThrowsDemo {
```

```
public void getKey() {

   Scanner myScanner = new Scanner( System.in );

   String key = myScanner.next();
   printDetails( key );

  }

  public void printDetails(String key) {

    try {  String message = getDetails(key);

           System.out.println( message );

          }catch ( Exception e){

            throw e;

          }

    }

  private String getDetails(String key) {

   if(key == "") {
       throw new Exception( "Key set to empty string" );

    }

  return "data for " + key; }

  }
```

Листинг 6

При попытке компиляции вы получите следующий синтаксис ошибки:

Unreported java.lang.Exception исключение;

В результате успешного пробрасывания исключение должен быть поймано или объявлено. Причиной полученной ошибки является выражение ***throw e.***

В данном случае метод printDetails () решил, что он не может иметь дело с исключением и проходит все дерево его вызовов. Поскольку метод GetKey() не имеет блока try-catch для обработки исключений, то Java становится перед выбором, что в таком случае делать.

Проблему можно решить несколькими способами:

1) Добавьте соответствующие try-catch блоки таким образом, чтобы в конечном итоге один из них обрабатывал исключение;
2) Удалите блоки try-catch для всех методов, кроме одного, который обрабатывает исключение. Добавьте throws, котрый бросает исключение методу, который проходит исключение без обработки.

Задание № 7

Измените код программы, чтобы она работала корректно.

Следующий фрагмент кода программы демонстрирует случай, когда никакие методы не будут брать на себя ответственность за обработку ошибок, и программа в конечном итоге выходит “сломается “.

```
public class ThrowsDemo {

   public void getKey() throws Exception {

   Scanner myScanner = new Scanner( System.in );

   System.out.print("Enter Key ");
   String key = myScanner.nextLine();
   printDetails( key );

   }
```

```
        public void printDetails(String key) throws Exception {

        String message = getDetails(key);

        System.out.println( message ); }

        private String getDetails(String key) throws Exception {

          if(key.equals("")) {
             throw new Exception( "Key set to empty string" );

           }

        return "data for " + key;

        }

}
```

Листинг 7

Задание №8

Измените этот код следующим образом:

1. Удалите throws Exception из метода getKey().
2. Измените метод getKey(), добавив try-catch блок для обработки исключений.
3. Добавьте цикл к getKey() таким образом, чтобы пользователь получил еще один шанс на ввод значение ключа

Замечания:

Инструкция throw очень аналогична инструкции return – она прекращает выполнение метода, дальше мы не идем. Если мы нигде не ставим catch, то у нас бросок Exception очень похож на System.exit() – система вырубается. Но мы в любом месте можем поставить catch и, таким образом, предотвратить поломку кода.

Выводы:

Фактически при работе с исключениями весь материал делится на две части: *синтаксис* (ответ на вопрос, что компилятор пропустит, а что нет) и *семантика* (вопрос, как лучше делать) исключений. В отличие от вариантов с for, while, switch, использование исключений – более сложный механизм. Но он сложен не синтаксически, а семантически, по своему подходу. То есть при генерации исключений нужно думать о том - не как правильно его использовать, а *с каким смыслом* его использовать. То есть вопрос, в каких ситуациях стоит ли ломать систему и где, а в каких ситуациях ее восстанавливать.

В хорошей инженерной системе каждый любой модуль всегда проверяет все аргументы на входе.

Можно еще сказать что существует иерархия различных способов *прекращения выполнения некоего действия (или ряда действий) в виде участка кода* и эта иерархия классифицирует возможные действия по мощности используемого оператора: continue, break, return, throw.

- **continue** прекращает выполнение *данной итерации* в цикле;
- **break** прекращает выполнение *данного блока (например цикла)*;
- **return** – это инструкция выхода *из данного метода (например прекращение выполнения функции)*;
- **throw** – еще более сильная инструкция, она прекращает выполнение данного метода *и метода, который его вызвал*. Так-как исключения вообще-то позволяют сломать весь стек работы программы.

ЛАБОРАТОРНАЯ РАБОТА №10 РАБОТА С ДЖЕНЕРИКАМИ. СТИРАНИЕ ТИПОВ

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:

Цель данной лабораторной работы – научиться работать с обобщенными типами в Java и применять их в программах.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:

Понятие дженериков.

Введение в Дженерики. В JDK представлены дженерики (перевод с англ. generics), которые поддерживают абстрагирование по типам (или параметризованным типам). В объявлении классов дженерики представлены обобщенными типами, в то время как пользователи классов могут быть конкретными типами, например во время создания объекта или вызова метода.

Вы, конечно, знакомы с передачей аргументов при вызове методов в языке C++. Когда вы передаете аргументы внутри круглых скобок () в момент вызова метода, то аргументы подставляются вместо формальных параметров с которыми объявлен и описан метод. Схожим образом в generics вместо передаваемых аргументов мы передаем информацию только о типах аргументов внутри угловых скобок <> (так называемая diamond notation или алмазная запись).

Основное назначение использования дженериков — это абстракция работы над типами при работе с коллекциями («Java Collection Framework»).

Например, класс ArrayList можно представить следующим образом для получения типа дженериков <E> следующим образом:

```
public class ArrayList<E> implements List<E> .... {

  // Constructor

  public ArrayList() { ..... }


  // Public methods

  public boolean add(E e) { ..... }

  public void add(int index, E element) { ..... }

  public boolean addAll(int index, Collection<? extends E> c)

  public abstract E get(int index) { ..... }

  public E remove(int index)

  .....

}
```

Чтобы создать экземпляр ArrayList, пользователям необходимо предоставить фактический тип для <E> для данного конкретного экземпляра. Фактический тип будет заменять все ссылки на E внутри класса. Например:

```
ArrayList<Integer> lst1 = new ArrayList<Integer>(); // E substituted with Integer

lst1.add(0, new Integer(88));

lst1.get(0);


ArrayList<String> lst2 = new ArrayList<String>();   // E substituted with String
```

```
        lst2.add(0, "Hello");
        lst2.get(0);
```

В приведенном выше примере показано, что при проектировании или определении классов, они могут быть типизированными по общему типу; в то время как пользователи классов предоставляют конкретную фактическую информацию о типе во время создания экземпляра объекта типа класс. Информация о типе передается внутри угловых скобок <>, так же как аргументы метода передаются внутри круглой скобки ().

В этом плане общие коллекции не являются безопасными типами!

Если вы знакомы с классами- коллекциями, например, такими как ArrayList, то вы знаете, что они предназначены для хранения объектов типа java.lang.Object. Используя основной принцип ООП-полиморфизм, любой подкласс класса Object может быть заменен на Object. Поскольку Object является общим корневым классом всех классов Java, то коллекция, предназначенная для хранения Object, может содержать любые классы Java. Однако есть одна проблема. Предположим, например, что вы хотите определить ArrayList из объектов класса String. То при выполнении операций, например операции add(Object) объект класса String будет неявным образом преобразовываться в Object компилятором. Тем не менее, во время поиска ответственность программиста заключается в том, чтобы явно отказаться от Object обратно в строку. Если вы непреднамеренно добавили объект не-String в коллекцию, то компилятор не сможет обнаружить ошибку, а понижающее приведение типов (от родителя к дочернему) завершится неудачно во время выполнения (будет сгенерировано ClassCastException throw). Ниже приведен пример:

```
import java.util.*;
public class ArrayListWithoutGenericsTest {
  public static void main(String[] args) {
    List strLst = new ArrayList();  // List and ArrayList holds Objects
    strLst.add("alpha");            // String upcast to Object implicitly
    strLst.add("beta");
    strLst.add("charlie");
    Iterator iter = strLst.iterator();
    while (iter.hasNext()) {
      String str = (String)iter.next(); // need to explicitly downcast Object back to String
      System.out.println(str);
    }
    strLst.add(new Integer(1234));       // Compiler/runtime cannot detect this error
    String str = (String)strLst.get(3);  // compile ok, but runtime ClassCastException
  }
}
```

Мы могли бы использовать оператор создания объектов типа класс для проверки правильного типа перед добавлением. Но опять же, при создании объектов (инстанцировании) проблема обнаруживается во время выполнения. Как насчет проверки типа во время компиляции?

Использование дженериков.

Давайте напишем наш собственный «безопасный тип» ArrayList

Мы проиллюстрируем использование дженериков путем написания нашего собственного типа изменяемого размера массива для хранения определенного типа объектов (аналогично ArrayList).

Начнем с версии MyArrayList без дженериков:

```
// A dynamically allocated array which holds a collection of java.lang.Object - without generics
public class MyArrayList {
  private int size;    // количество элементов – размер коллекции
  private Object[] elements;

  public MyArrayList() {        //конструктор
    elements = new Object[10];  // начальная инициализация емкостью 10 элементов
    size = 0;
  }

  public void add(Object o) {
    if (size < elements.length) {
      elements[size] = o;
    } else {
      // выделить массив большего размера и добавить элемент,
    }
    ++size;
  }

  public Object get(int index) {
    if (index >= size)
      throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
    return elements[index];
  }
  public int size() { return size; }
}
```

В данном примере класс MyArrayList не является безопасным типом. Например, если мы создаем MyArrayList, который предназначен для хранения только String, но допустим в процессе работы с ним добавляется Integer. Что произойдет? Компилятор не сможет обнаружить ошибку. Это связано с тем, что MyArrayList предназначен для хранения объектов Object, и любые классы Java являются производными от Object.

```
public class MyArrayListTest {
  public static void main(String[] args) {
    // Intends to hold a list of Strings, but not type-safe
    MyArrayList strLst = new MyArrayList();
    // добавление элементов строк (типа String) –это повышающее или
//расширяющее преобразование (upcasting) к типу Object
```

```
    strLst.add("alpha");
    strLst.add("beta");
    // при получении – необходимо явное понижающее преобразование (downcasting) назад к String
    for (int i = 0; i < strLst.size(); ++i) {
      String str = (String)strLst.get(i);
      System.out.println(str);
    }

    // случайно добавленный не-String объект, произойдет вызов во время выполнения
    // ClassCastException. Компилятор не может отловить ошибку.
    strLst.add(new Integer(1234));   // компиляция/выполнение - не можем обнаружить эту ошибку
    for (int i = 0; i < strLst.size(); ++i) {
      String str = (String)strLst.get(i);  // компиляция ok,  при выполнении (runtime) ClassCastException
      System.out.println(str);
    }
  }
}
```

Если вы намереваетесь создать список объектов String, но непреднамеренно добавленный в этот список не-String-объекты будут преобразованы к типу Object. Компилятор не сможет проверить, является ли понижающее преобразование типов действительным во время компиляции (это известно как позднее связывание или динамическое связывание). Неправильное понижающее преобразование типов будет выявлено только во время выполнения программы, в виде исключения ClassCastException, а это слишком поздно, для того чтобы внести изменения в код и исправить работу программы. Компилятор не сможет поймать эту ошибку в момент компиляции. Вопрос в том, как заставить компилятор поймать эту ошибку и обеспечить безопасность использования типа во время выполнения.

Классы дженерики или параметризованные классы

В JDK введены так называемые обобщенные или параметризованные типы – generics или по-другому обобщенные типы для решения вышеописанной проблемы. Параметризованных (generic) классы и методы, позволяют использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Использование параметризации позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Дженерики или обобщенные типы позволяют вам ***абстрагироваться*** от использования конкретных типов. Вы можете создать класс с таким общим типом и предоставить информацию об определенном типе во время создания экземпляра объекта типа класс. А компилятор сможет выполнить необходимую проверку типов во время компиляции. Таким образом, вы сможете убедиться, что во время выполнения программы не возникнет ошибка выбора типа еще на этапе компиляции, что как раз является безопасностью для используемого типа.

Рассмотрим пример: объявление интерфейса java.util.List <E>:

```
public interface List<E> extends Collection<E> {
  boolean add(E o);
  void add(int index, E element);
  boolean addAll(Collection<? extends E> c);
```

```
  boolean containsAll(Collection<?> c);

  .....

}
```

Такая запись - <E> называется формальным параметром типа, который может использоваться для передачи параметров «типа» во время создания фактического экземпляра типа. Механизм похож на вызов метода Напомним, что в определении метода мы всегда объявляем формальные параметры для передачи данных в метод (при описании метода используются формальные параметры, а при вызове на их место подставляются аргументы). Например как представлено ниже:

```
// Определение метода

public static int max(int a, int b) {  // где int a, int b это формальные параметры

  return (a > b) ? a : b;

}
```

А во время вызова метода формальные параметры заменяются фактическими параметрами (аргументами). Например так:

```
// Вызов: формальные параметры, замененные фактическими параметрами
int maximum = max(55, 66);   // 55 и 66 теперь фактические параметры

int a = 77, b = 88;

maximum = max(a, b);         // a и b теперь фактические параметры
```

Параметры формального типа, используемые в объявлении класса, имеют ту же цель, что и формальные параметры, используемые в объявлении метода. Класс может использовать формальные параметры типа для получения информации о типе, когда экземпляр создается для этого класса. Фактические типы, используемые во время создания, называются фактическими типами параметров.

Вернемся к java. util.List <E>, итак в действительности, когда тип определен, например List <Integer>, все вхождения параметра формального типа <E> заменяются актуальным или фактическим параметром типа <Integer>. Используя эту дополнительную информацию о типе, компилятор может выполнить проверку типа во время компиляции и убедиться, что во время выполнения не будет ошибки при использовании типов.

Конвенция кода Java об именах для формальных типов

Мы должны помнить, что написания “чистого кода” необходимо руководствоваться конвенцией кода на Java. Поэтому, для создания имен формальных типов используйте один и тот же символ в верхнем регистре. Например,

- <E> для элемента коллекции;
- <T> для обобщенного типа;
- <K, V> ключ и значение.
- <N> для чисел
- S,U,V, и т.д. для второго, третьего, четвертого типа параметра

Рассмотрим пример параметризованного или обобщенного типа как класса. В нашем примере класс GenericBox принимает общий параметр типа E, содержит содержимое типа E. Конструктор, геттер и сеттер работают с параметризованным типом E. Метод нашего класса toString() демонстрирует фактический тип содержимого.

```
public class GenericBox<E> {

  // Private переменная класса

  private E content;


  // конструктор

  public GenericBox(E content) {
```

```
    this.content = content;
  }

  public E getContent() {
    return content;
  }

  public void setContent(E content) {
    this.content = content;
  }

  public String toString() {
    return content + " (" + content.getClass() + ")";
  }
}
```

В следующем примере мы создаем GenericBoxes с различными типами (String, Integer и Double). Обратите внимание, что при преобразовании типов происходит автоматическая автоупаковка и распаковка для преобразования между примитивами и объектами-оболочками (мы с вами это изучали ранее).

```
public class TestGenericBox {
  public static void main(String[] args) {
    GenericBox<String> box1 = new GenericBox<String>("Hello");
    String str = box1.getContent(); // явного понижающего преобразования (downcasting) не требуется
    System.out.println(box1);
    GenericBox<Integer> box2 = new GenericBox<Integer>(123);  //автоупаковка типа int в тип Integer
    int i = box2.getContent();       // (downcast) понижающее преобр. к типу Integer, автоупаковка в тип int
    System.out.println(box2);
    GenericBox<Double> box3 = new GenericBox<Double>(55.66);  ////автоупаковка типа double в тип  Double
    double d = box3.getContent();     // (downcast) понижающее преобр.  к типу Double,
//автоупаковка в тип double
    System.out.println(box3);
  }
}
```

Вывод программы:
Hello (class java.lang.String)
123 (class java.lang.Integer)
55.66 (class java.lang.Double)

Стирание типов

Из предыдущего примера может создаться видимость того, что компилятор заменяет параметризованный тип <E> актуальным или фактическим типом (таким как String, Integer) во время создания экземпляра объекта типа класс. Если это так, то компилятору необходимо будет создавать новый класс для каждого актуального или фактического типа (аналогично шаблону C ++).

На самом же деле происходит следующее - компилятор заменяет всю ссылку на параметризованный тип E на ссылку на Object, выполняет проверку типа и вставляет требуемые операторы, обеспечивающие понижающее приведения типов. Например, GenericBox компилируется следующим образом (который совместим с кодами без дженериков):

```
public class GenericBox {

  // Private переменная

  private Object content;


  // Constructor

  public GenericBox(Object content) {

    this.content = content;

  }


  public Object getContent() {

    return content;

  }


  public void setContent(Object content) {

    this.content = content;

  }


  public String toString() {

    return content + " (" + content.getClass() + ")";

  }

}
```

Компилятор сам вставляет требуемый оператор понижения типа (downcasting) в код:

```
GenericBox box1 = new GenericBox("Hello");  // upcast безопасно для типов
String str = (String)box1.getContent();     // компилятор вставляет операцию понижения типа (downcast)
System.out.println(box1);
```

Вывод. Таким образом, для всех типов используется одно и то же определение класса. Самое главное, что байт-код всегда совместим с теми классами, у которых нет дженериков. Этот процесс называется *стиранием типа.*

Рассмотрим операцию “стирания типов” с помощью нашего «безопасного типа» ArrayList, который мы рассматривали ранее в паримере.

Вернемся, к примеру MyArrayList. С использованием дженериков мы можем переписать нашу программу как показано в листинге ниже:

```
// Динамически выделяемая память для массива с дженериками
public class MyGenericArrayList<E> {
  private int size;     // количество элементов- емкость списка
  private Object[] elements;

  public MyGenericArrayList() {  //конструктор
    elements = new Object[10];  // выделяем память сразу для 10 элементов списка при его создании
    size = 0;
  }

  public void add(E e) {
    if (size < elements.length) {
      elements[size] = e;
    } else {
// добавляем элемент к списку и увеличиваем счетчик количества элементов

    }
    ++size;
  }

  public E get(int index) {
    if (index >= size)
      throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
    return (E)elements[index];
  }

  public int size() { return size; }
}
```

В объявлении MyGenericArrayList <E> объявляется *класс-дженерик* с формальным параметром типа <E>. Во время фактического создания объектов типа класс, например, MyGenericArrayList <String>, определенного типа <String> или параметра фактического типа, подставляется вместо параметра формального типа <E>.

Помните! Что Дженерики реализуются компилятором Java в качестве интерфейсного преобразования, называемого стиранием, которое переводит или перезаписывает код, который использует дженерики в не обобщенный код (для обеспечения обратной совместимости). Это преобразование стирает

всю информацию об общем типа. Например, ArrayList <Integer> станет ArrayList. Параметр формального типа, такой как <E>, заменяется объектом по умолчанию (или верхней границей типа). Когда результирующий код не корректен, компилятор вставляет оператор преобразования типа.

Следовательно, код, транслированный компилятором, т о есть переведенный код выглядит следующим образом:

```
// The translated code

public class  MyGenericArrayList {

  private int size;     // количество элементов

  private Object[] elements;


  public MyGenericArrayList() {  // конструктор

    elements = new Object[10];  // выделяем память для первых 10 объектов

    size = 0;

  }


  // Компилятор заменяет параметризованный тип E на Object, но проверяет, что параметр e имеет тип E, когда //он используется для обеспечения безопасности типа


  public void add(Object e) {

    if (size < elements.length) {

      elements[size] = e;

    } else {

      // allocate a larger array and add the element, omitted

    }

    ++size;

  }


  // Компилятор заменяет E на Object и вводит оператор понижающего преобразования типов (E <E>) для //типа возвращаемого значения при вызове метода


  public Object get(int index) {

    if (index >= size)

      throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);

    return (Object)elements[index];

  }


  public int size() {

    return size;

  }
```

```
}
```

Когда класс создается с использованием актуального или фактического параметра типа, например. MyGenericArrayList <String>, компилятор гарантирует, что add(E e) работает только с типом String. Он также вставляет соответствующий оператор понижающее преобразование типов в соответствие с типом возвращаемого значения E для метода get(). Например,

```
public class MyGenericArrayListTest {

  public static void main(String[] args) {

    // безопасный тип для хранения списка объектов Strings (строк)

    MyGenericArrayList<String>  strLst = new MyGenericArrayList <String>();


    strLst.add("alpha");   // здесь компилятор проверяет, является ли аргумент типом String

    strLst.add("beta");


    for (int i = 0; i < strLst.size(); ++i) {

      String str = strLst.get(i); //компилятор вставляет оператор

//понижающего преобразования operator (String)

      System.out.println(str);

    }


    strLst.add(new Integer(1234));  // компилятор обнаруживает аргумент,

//который не является объектом String, происходит ошибка компиляции

  }

  }
```

Выводы: с помощью дженериков компилятор может выполнять проверку типов во время компиляции и обеспечивать безопасность использования типов во время выполнения.

Запомните. В отличие от «шаблона» в C ++, который создает новый тип для каждого определенного параметризованного типа, в Java класс generics компилируется только один раз, и есть только один файл класса, который используется для создания экземпляров для всех конкретных типов.

Обобщенные методы (параметризованные методы)

Методы также могут быть определены с помощью общих типов (аналогично родовому классу). Например,

```
public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {

  for (E e : a) lst.add(e);

}
```

При объявлении обобщенного метода или метода-дженерика могут объявляться параметры формального типа (например, такие как <E>, <K, V>) перед возвращаемым типом (в примере выше это static <E> void). Параметры формального типа могут затем использоваться в качестве заполнителей для типа возвращаемого значения, параметров метода и локальных переменных в общем методе для правильной проверки типов компилятором.

Подобно классам-дженерикам, когда компилятор переводит общий метод, он заменяет параметры формального типа, используя операцию стирания типа. Все типы заменяются типом Object по умолчанию

(или верхней границей типа – типом классом, стоящим на самой вершине иерархии наследования). Переведенная на язык компилятора версия программы выглядит следующим образом:

```
public static void ArrayToArrayList(Object[] a, ArrayList lst) {// компилятор проверяет, есть ли тип E[],
                                                                 // lst имеет тип ArrayList<E>
 ArrayList<E>
  for (Object e : a) lst.add(e);                    // компилятор проверяет является ли e типом E
}
```

Компилятор всегда проверяет, что a имеет тип E[], lst имеет тип ArrayList <E>, а e имеет тип E во время вызова для обеспечения безопасности типа. Например,

```
import java.util.*;
public class TestGenericMethod {


  public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
    for (E e : a) lst.add(e);
  }


  public static void main(String[] args) {
    ArrayList<Integer> lst = new ArrayList<Integer>();


    Integer[] intArray = {55, 66};  // автоупаковка
    ArrayToArrayList(intArray, lst);
    for (Integer i : lst) System.out.println(i);


    String[] strArray = {"one", "two", "three"};
    //ArrayToArrayList(strArray, lst);   //ошибка компиляции ниже
  }
}
```

```
TestGenericMethod.java:16: <E>ArrayToArrayList(E[],java.util.ArrayList<E>) in TestGenericMethod
cannot be applied to (java.lang.String[],java.util.ArrayList<java.lang.Integer>)
    ArrayToArrayList(strArray, lst);
    ^
```

У дженериков есть необязательный синтаксис для указания типа для общего метода. Вы можете поместить фактический тип в угловые скобки <> между оператором точки и именем метода. Например,

```
TestGenericMethod.<Integer>ArrayToArrayList(intArray, lst);
```

Обратите внимание на точку после имени метода

Подстановочный синтаксис в Java (WILD CARD)

Одним из наиболее сложных аспектов generic-типов (обобщенных типов) в языке Java являются wildcards (подстановочные символы, в данном случае – «?»), и особенно – толкование и разбор запутанных сообщений об ошибках, происходящих при wildcard capture (подстановке вычисляемого компилятором типа вместо wildcard). В своем труде «Теория и практика Java» (Java theory and practice) старейший Java-разработчик Брайен Гетц расшифровывает некоторые из наиболее загадочно выглядящих сообщений об ошибках, выдаваемых компилятором «javac», и предлагает решения и варианты обхода, которые помогут упростить использование generic-типов.

Рассмотрим следующую строку кода:

```
ArrayList<Object> lst = new ArrayList<String>();
```

Компиляция вызовет ошибку - «несовместимые типы», поскольку ArrayList <String> не является ArrayList <Object>. Эта ошибка противоречит нашей интуиции в отношении полиморфизма, поскольку мы часто присваиваем экземпляр подкласса ссылке на суперкласс.

Рассмотрим эти два утверждения:

```
List<String> strLst = new ArrayList<String>();   // строчка 1
List<Object> objLst = strList;                    // строчка 2 – ошибка компиляции
```

Выполнение строки 2 генерирует ошибку компиляции. Но если строка 2 выполняется, то некоторые объекты добавляются в objLst, а strLst будут «повреждены» и больше не будет содержать только строки. (так-как переменные objLst и strLst содержат одинаковую ссылку или ссылаются на одну и ту-же область памяти).

Учитывая вышеизложенное, предположим, что мы хотим написать метод printList (List <.>) Для печати элементов списка. Если мы определяем метод как printList (List <Object> lst), он может принимать только аргумент List <object>, но не List <String> или List <Integer>. Например,

```
import java.util.*;
public class TestGenericWildcard {


  public static void printList(List<Object> lst) {  // accept List of Objects only,
                                                     // not List of subclasses of object
    for (Object o : lst) System.out.println(o);
  }


  public static void main(String[] args) {
    List<Object> objLst = new ArrayList<Object>();
    objLst.add(new Integer(55));
    printList(objLst);   // matches


    List<String> strLst = new ArrayList<String>();
    strLst.add("one");
    printList(strLst);  // compilation error
```

```
    }
}
```

Использование подстановочного знака без ограничений в описании типа <?>

Чтобы разрешить проблему, описанную выше, необходимо использовать подстановочный знак (?), он используется в дженериках для обозначения любого неизвестного типа. Например, мы можем переписать наш printList () следующим образом, чтобы можно было передавать список любого неизвестного типа.

```
public static void printList(List<?> lst) {
  for (Object o : lst) System.out.println(o);
}
```

Использование подстановочного знака в начале записи типа <? extends тип>

Подстановочный знак <? extends type> обозначает тип и его подтип. Например,

```
public static void printList(List<? extends Number> lst) {
  for (Object o : lst) System.out.println(o);
}
```

List<? extends Number> принимает список Number и любой подтипNumber, например, List <Integer> и List <Double>. Понятно, что обозначение типа <?> можно интерпретировать как <? extends Object>, который применим ко всем классам Java.
Другой пример,

```
//List<Number> lst = new ArrayList<Integer>();  // ошибка компиляции
List<? extends Number> lst = new ArrayList<Integer>();
```

Задачи

1. Написать метод для конвертации массива строк/чисел в список.
2. Написать класс, который умеет хранить в себе массив любых типов данных (int, long etc.).
3. Реализовать метод, который возвращает любой элемент массива по индексу.
4. Написать функцию, которая сохранит содержимое каталога в список и выведет первые 5 элементов на экран.
5. *Реализуйте вспомогательные методы в классе Solution, которые должны создавать соответствующую коллекцию и помещать туда переданные объекты. Методы newArrayList, newHashSet параметризируйте общим типом T. Метод newHashMap параметризируйте парой <K, V>, то есть типами K- ключ и V-значение. Аргументы метода newHashMap должны принимать

ССЫЛКИ ДЛЯ ЧТЕНИЯ

1. http://java-s-bubnom.blogspot.com/2015/07/generic.html
2. http://www.quizful.net/post/java-generics-tutorial
3. http://crypto.pp.ua/2010/06/parametrizovannye-klassy-java/
4. https://habr.com/post/207360/
5. https://habr.com/ru/company/sberbank/blog/416413/
6. https://www.geeksforgeeks.org/wildcards-in-java/
7. https://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html
8. https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html