Generic — продолжаем путешествие

Generic — предки и наследники

В предыдущей главе Коллекции – базовые принципы мы написали пример для сортировки, где были вынуждены создать ДВА метода для печати коллекции. Думаю, что вы уже оценили это как не самый лучший вариант — совершенно с вами согласен. Давайте несколько модифицируем наш пример и на его основе узнаем новые возможности generic. Создадим очень простой класс с одним полем — **BasicClass**

- Расширенное описание классов
- Исключения
- Решения на основе классов
- Список контактов — начало
- Коллекции — базовые принципы
- Коллекции — продолжение
- Список контактов — GUI приложение
- Что такое JAR-файлы
- Многопоточность — первые шаги
- Многопоточность и синхронизация
- Работаем с XML
- Reflection — основы
- Установка СУБД PostgreSQL
- Базы данных на Java — первые шаги
- Возможности JDBC — второй этап
- JDBC — групповые операции
- Список контактов — работаем с БД
- Переезжаем на Maven
- Потоки ввода-вывода
- Сетевое взаимодействие
- С чего начинается Web

```
1  package edu.javacourse.collection;
2
3  public class BasicClass
4  {
5      private String name;
6
7      public BasicClass(String name) {
8          this.name = name;
9      }
10
11     public String getName() {
12         return name;
13     }
14 }
```

И двух наследников этого класса — **MyClass1** и **MyClass2**.

```java
package edu.javacourse.collection;

public class MyClass1 extends BasicClass
{
    public MyClass1(String name) {
        super(name);
    }
}
```

```java
package edu.javacourse.collection;

public class MyClass2 extends BasicClass
{
    public MyClass2(String name) {
        super(name);
    }
}
```

Теперь сделаем очень простой пример, в котором заполним коллекции из двух классов-наследников и сделаем метод, который печатает поле **name** класса **BasicClass**. Думаю, что это несложно и должно быть понятно.

```java
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.List;

public class Main
{
    public static void main(String[] args) {
        List<MyClass1> list1 = new ArrayList<MyClass1>();
        list1.add(new MyClass1("Василий 1"));
        list1.add(new MyClass1("Павел 1"));
        list1.add(new MyClass1("Андрей 1"));
        list1.add(new MyClass1("Петр 1"));
        list1.add(new MyClass1("Анжелика 1"));
        printCollection("Коллеция 1", list1);

        List<MyClass2> list2 = new ArrayList<MyClass2>();
        list2.add(new MyClass2("Василий 2"));
        list2.add(new MyClass2("Павел 2"));
        list2.add(new MyClass2("Андрей 2"));
```

```
        list2.add(new MyClass2("Петр 2"));
        list2.add(new MyClass2("Анжелика 2"));
        printCollection("Коллекция 2", list2);

    }

    private static void printCollection(String title, List list) {
        System.out.println(title);
        for (Object mc : list) {
            // Т.к. классы - наследники BasicClass, то обе коллекции
            // содержат объекты этого типа.
            BasicClass bc = (BasicClass)mc;
            System.out.println("Item:" + bc.getName());
        }
        System.out.println();
    }
}
```

Что здесь важно отметить — в метод **printCollection** мы в качестве аргумента передаем список и (ЧТО ВАЖНО) мы не указали тип элементов. Это нехорошо, т.к. мы можем передать тогда ЛЮБУЮ коллекцию в метод **printCollection**, хотя рассчитываем ТОЛЬКО на наследников **BasicClass**. Что же делать ? Первое решение, которое приходит в голову — воспользоваться полиморфизмом и сделать так:

```
    private static void printCollection(String title, List<BasicClass> list) {
        System.out.println(title);
        for (BasicClass bc : list) {
            System.out.println("Item:" + bc.getName());
        }
        System.out.println();
    }
```

Очень даже красиво, но что мы видим выше метода ? Ошибка. Компилятор отказывается принимать коллекции, которые типизированы наследниками класса **BasicClass**. Вот так незадача. Оказывается, generic не подчиняется законам полиморфизма. Т.к. у нас пример простой, то мы можем попытаться перехитрить компилятор и создавать коллекции, которые типизируются классом **BasicClass**.

```
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.List;

public class Main
```

```
{
    public static void main(String[] args) {
        List<BasicClass> list1 = new ArrayList<BasicClass>();
        list1.add(new MyClass1("Василий 1"));
        list1.add(new MyClass1("Павел 1"));
        list1.add(new MyClass1("Андрей 1"));
        list1.add(new MyClass1("Петр 1"));
        list1.add(new MyClass1("Анжелика 1"));
        printCollection("Коллеция 1", list1);

        List<BasicClass> list2 = new ArrayList<BasicClass>();
        list2.add(new MyClass2("Василий 2"));
        list2.add(new MyClass2("Павел 2"));
        list2.add(new MyClass2("Андрей 2"));
        list2.add(new MyClass2("Петр 2"));
        list2.add(new MyClass2("Анжелика 2"));
        printCollection("Коллекция 2", list2);
    }

    private static void printCollection(String title, List<BasicClass> list) {
        System.out.println(title);
        for (BasicClass bc : list) {
            System.out.println("Item:" + bc.getName());
        }
        System.out.println();
    }
}
```

Это работает. Но мы можем использовать **list1** и **list2** как коллекцию элементов класса **BasicClass**, а нам надо эти коллекции различать — одна для класса **MyClass1**, другая — для **MyClass2**. Иначе это не комильфо. Ну сами посудите — мы по сути приближаемся к варианту без generic вообще. Если почитать документацию, то можно найти вот такое решение:

```
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.List;

public class Main
{
    public static void main(String[] args) {
        List<MyClass1> list1 = new ArrayList<>();
        list1.add(new MyClass1("Василий 1"));
        list1.add(new MyClass1("Павел 1"));
        list1.add(new MyClass1("Андрей 1"));
```

```
        list1.add(new MyClass1("Петр 1"));
        list1.add(new MyClass1("Анжелика 1"));
        printCollection("Коллеция 1", list1);

        List<MyClass2> list2 = new ArrayList<>();
        list2.add(new MyClass2("Василий 2"));
        list2.add(new MyClass2("Павел 2"));
        list2.add(new MyClass2("Андрей 2"));
        list2.add(new MyClass2("Петр 2"));
        list2.add(new MyClass2("Анжелика 2"));
        printCollection("Коллекция 2", list2);
    }

    private static void printCollection(String title, List<?> list) {
        System.out.println(title);
        for (Object mc : list) {
            // Т.к. классы - наследники BasicClass, то обе коллекции
            // содержат объекты этого типа.
            BasicClass bc = (BasicClass)mc;
            System.out.println("Item:" + bc.getName());
        }
        System.out.println();
    }
}
```

Посмотртие внимательно на объявление метода **printCollection** — мы подставили знак «?» в угловые скобки. И наш код стал компилироваться. Знак «?» позволяет сказать, что мы готовы принимать любой класс внутри нашего списка. Это конечно прогресс, но как-то не совсем радует — этот вариант мало отличается от полностью нетипизированной коллекции. В общем тоже не так, как реально хочется. А хочется научить компилятор понимать, что мы хотим передавать в метод **printCollection** список, элементы которого являются НАСЛЕДНИКАМИ **BasicClass**. И generic предлагает такое решение. Смотрим новый вариант и сосредоточим наше внимание опять на объявление метода **printCollection**.

```
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.List;

public class Main
{
    public static void main(String[] args) {
        List<MyClass1> list1 = new ArrayList<>();
        list1.add(new MyClass1("Василий 1"));
        list1.add(new MyClass1("Павел 1"));
        list1.add(new MyClass1("Андрей 1"));
```

```
        list1.add(new MyClass1("Петр 1"));
        list1.add(new MyClass1("Анжелика 1"));
        printCollection("Коллеция 1", list1);

        List<MyClass2> list2 = new ArrayList<>();
        list2.add(new MyClass2("Василий 2"));
        list2.add(new MyClass2("Павел 2"));
        list2.add(new MyClass2("Андрей 2"));
        list2.add(new MyClass2("Петр 2"));
        list2.add(new MyClass2("Анжелика 2"));
        printCollection("Коллекция 2", list2);
    }

    private static void printCollection(String title, List<? extends BasicClass> list) {
        System.out.println(title);
        for (BasicClass bc : list) {
            System.out.println("Item:" + bc.getName());
        }
        System.out.println();
    }
}
```

Вот оно, самое красивое и элегантное — мы принимаем не просто коллекцию, а коллекцию классов, которые являются наследниками класса **BasicClass**. В этом нам помогает вот такая конструкция:

```
List<? extends BasicClass> list
```

Наш метод принимает список, элементы которого являются наследниками класса **BasicClass**. То, что нам надо. С одной стороны мы можем передавать наследников (до любого колена), с другой — у нас есть ограничение. Я не смогу передать в метод **printCollection** список с элементами типа **String** или **Integer**. Более точные правила, более жесткие контракты — меньше ошибок в большом и сложном проекте. Есть контакт.

Мы познакомились с вариантом типизации с огрначинем базового класса. Но существует обратный вариант ограничения — когда вы сможете передавать с коллекцию только те классы, которые являются ПРЕДКАМИ. Я такую стуацию за всю свою практику пока не встречал. Но возможно вам она будет полезна. Так что не буду подробно рассказывать об этой конструкции — просто предлагаю в качестве примера посмотреть код:

```
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.List;

```

```
public class Main
{
    public static void main(String[] args) {
        List<BasicClass> list1 = new ArrayList<>();
        list1.add(new MyClass1("Василий 1"));
        list1.add(new MyClass1("Павел 1"));
        list1.add(new MyClass1("Андрей 1"));
        list1.add(new MyClass1("Петр 1"));
        list1.add(new MyClass1("Анжелика 1"));
        printCollection("Коллеция 1", list1);

        List<MyClass2> list2 = new ArrayList<>();
        list2.add(new MyClass2("Василий 2"));
        list2.add(new MyClass2("Павел 2"));
        list2.add(new MyClass2("Андрей 2"));
        list2.add(new MyClass2("Петр 2"));
        list2.add(new MyClass2("Анжелика 2"));
        printCollection("Коллекция 2", list2);
    }

    private static void printCollection(String title, List<? super MyClass2> list) {
        System.out.println(title);
        for (Object mc : list) {
            // Т.к. классы - наследники BasicClass, то обе коллекции
            // содержат объекты этого типа.
            BasicClass bc = (BasicClass)mc;
            System.out.println("Item:" + bc.getName());
        }
        System.out.println();
    }
}
```

Теперь в метод **printCollection** можно передавать коллекцию элементов, которые являются ПРЕДКАМИ класса **MyClass2** (ну или им самим). Обратите внимание, как мне пришлось изменить типизацию переменой **list1** — теперь там предок. Если вы попробуете использовать класс **MyClass1** — у вас будет ошибка компиляции.

Generic — типизируем методы

Я не думаю, что вы часто будете сталкиваться с такой конструкцией в самом начале своей профессиональной деятельности, но не сказать о ней я не могу (да и не хочу). Эта конструкция на самом деле встречается не так уж редко — например, это касается такого достаточно известного пакета как **Spring Framework**.
Штука в том, что вы можете типизировать не только классы, но и ОТДЕЛЬНЫЕ МЕТОДЫ. Я не уверен, что сейчас смогу вас убедить, что это весьма удобно, но саму конструкцию продемонстрирую.

```java
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class GenericMethod {

    // Объявление метода, который типизируется
    public <T> T addAndReturn(T element, Collection<T> collection) {
        collection.add(element);
        return element;
    }

    public static void main(String[] arg) {
        GenericMethod gm = new GenericMethod();

        // Вызываем метод для элемента типа String
        String stringElement = "stringElement";
        List<String> stringList = new ArrayList<>();
        String theElement1 = gm.addAndReturn(stringElement, stringList);
        System.out.println("Element1:" + theElement1);

        // Вызываем метод для элемента типа Integer
        Integer integerElement = 123;
        List<Integer> integerList = new ArrayList<>();
        Integer theElement2 = gm.addAndReturn(integerElement, integerList);
        System.out.println("Element2:" + theElement2);
    }
}
```

В этом примере вы можете видеть два момента:

1. Объявление типизированного метода. Вот такое вот хитрое объявление. После слова **public** вы указываете, что метод типизирован — опять наш символ «T» в угловых скобках (надеюсь, вы помните, что может быть и другое название). После него стоит просто «T» — это уже говорит, что мы возвращаем объект типа «T». Аргументы тоже типизированы — элемент и коллекция.
2. Пример вызова нашего метода для разных типов — **String** и **Integer**.

Вы можете спросить — так в чем же тут весь цимес ? В типизации метода «на лету». Я передаю в метод строку — и все внутри метода начинает работать со строкой. Передаю число — и тот же самый метод работает теперь с числом. Причем это только метод и никаких классов. Идея такая же, как и с типизацией классов, но локальнее — только один метод. Лаконично и удобно.

Коллекции — больше подробностей

В этом разделе мы попробуем посмотреть более подробно на некоторые классы, о которых говорили раньше. Будем рассматривать их в разрезе интерфейсов **List**, **Set** и **Map**. Давайте и начнем.

Интерфейс List

Как я уже говорил, самыми главными особенностями этого интерфейса являются следующие:

- Однозначный порядок элементов — в каком порядке вставляли, в таком они и будут
- Возможность добавлять одинаковые объекты
- Доступ к элементу на определенной позиции
- Возможность перемещаться по списку как от головы к хвосту, так и от хвоста к голове

В реальных проектах коллекции типа **List** используются очень часто, так что постарайтесь как можно быстрее к ним «привыкнуть» — эти классы должны стать для вас рабочими лошадками, которые надо уметь использовать.
Продемонстрируем на примере те функциональные возможности, которые предлагает интерфейс **List**

```
1  package edu.javacourse.collection;
2
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.List;
6  import java.util.ListIterator;
7
8  public class ListExample
9  {
10     public static void main(String[] args) {
11         // Создаем список для примера
12         List<String> test = new ArrayList<>();
13         for (int i = 1; i < 6; i++) {
14             test.add("Строка " + i);
15         }
16
17         // Стандартный проход по всем элементам коллекции
18         System.out.println();
19         System.out.println("Печать через цикл foreach");
20         for(String s : test) {
21             System.out.println(s);
22         }
23
```

```
        // Использование итератора
        System.out.println();
        System.out.println("Печать через итератор");
        for(Iterator<String> it = test.iterator(); it.hasNext(); ) {
            String s = it.next();
            System.out.println(s);
        }

        // Используя ListIterator можем двигаться в обоих направлениях
        System.out.println();
        System.out.println("Печать через итератор списка от конца к началу");
        for( ListIterator<String> li = test.listIterator(test.size()); li.hasPrevious(); ) {
            String s = li.previous();
            System.out.println(s);
        }
        System.out.println("Печать через итератор списка от начала к концу");
        for( ListIterator<String> li = test.listIterator(); li.hasNext(); ) {
            String s = li.next();
            System.out.println(s);
        }

        // Обращение к элементу через индекс (позицию)
        System.out.println();
        System.out.println("Печать через индекс элемента");
        for(int i=0; i<test.size(); i++) {
            String s = test.get(i);
            System.out.println(s);
        }
    }
}
```

В качестве самых распространенных классов, которые реализуют интерфейс **List** я бы назвал следующие: **java.utilArrayList, java.util.Vector, java.util.LinkedList**. Первый и второй классы основаны на массивах и имеют одинаковые достоинства и недостатки — быстрый доступ к эдементу по индексу и не самый быстрый алгоритм при добавлении новых элементов. **java.util.Vector** является потокобезопасным — об этом мы будем говорить в дальнейшем. **java.util.LinkedList** — это связный список, который позволяет быстро добавлять, но поиск по индексу зависит от размера коллекции.
Также я бы обратил ваше внимание на класс **java.util.Stack** — реализация стека (LIFO). В некоторых задачах бывает очень удобно использовать такую структуру.

Интерфейс Set и SortedList

В отличии от интерфейса **java.util.List** эти интерфейс **java.util.Set** не гарантирует вам порядок и не позволяет вставить одинаковые элементы в коллекцию. Наиболее популярными классами на мой взгляд являются следующие:

- **HashSet** — коллекция позволяет дотаточно быстро получить доступ к объекту по хеш-коду
- **TreeSet** — это наследник интерфейса **SortedSet**. И этот интерфейс сортирует все элементы и накладывает на них ограничение — они должны реализовать интерфейс **Comparable** — мы уже с ним сталкивались, когда говорили о сортировке
- **LinkedHashSet** — достаточно любопытный класс, который СОХРАНЯЕТ порядок вставленных элементов и НЕ МЕНЯЕТ их порядок, если элемент вставляется повторно.

Можете самостоятельно разобрать пример

```
1  package edu.javacourse.collection;
2
3  import java.util.LinkedHashSet;
4  import java.util.Set;
5
6  public class ListExample
7  {
8      public static void main(String[] args) {
9          Set<String> test = new LinkedHashSet<>();
10         // Заполняем от 5 до 1
11         for (int i = 5; i > 0; i--) {
12             test.add("Строка " + i);
13         }
14         for (String s : test) {
15             System.out.println(s);
16         }
17         System.out.println();
18
19         // Заполняем (заменяем) от 1 до 5
20         for (int i = 1; i < 6; i++) {
21             test.add("Строка " + i);
22         }
23         for (String s : test) {
24             System.out.println(s);
25         }
26         System.out.println();
27     }
28 }
```

Как видите, мы сначала заполняем элементы от «Строка 5» до «Строка 1», а потом делаем повторную вставку, но в другом порядке. Порядок элементов в самой коллекции при этом не меняется. Попробуйте сделать наоборот — сначала заполнить коллекцию элементами от «Строка 1» до «Строка 5» — посмотрите, что получится.

Интерфейс Map

Самое главное в этом интерфейсе (как и в любом другом) выделить самое главное его предназначение. В данном случае это возможность получить доступ к нужному объекту из множества объектов по ключу. Например, при регистрации на сайте у вас запрашивают e-mail, который часто является логином. Также вы можете ввести некоторую дополнительную информацию — имя, фамилию, какие-то данные о дате рождения, области ваших интересов и прочая. Но ключом (и в прямом и переносном смыслах) ко всему этому пакету информации является ваш e-mail/логин.
Если у вам необходимо работать с группой информационных блоков с таким свойством, то **Map** явлется очень удобной структурой. По сути вы можете всегда спросить у такого класса: «дай объект по такому ключу». И если такой объект есть — вам отдадут нужный объект. Причем важно отметить следующее — ключ в пределах коллекции должен быть уникальным. Хотя есть реализации других разработчиков, которые это ограничение обходят. А вот объект под разными ключами может регистрироваться несколько раз.
В итоге, наиболее интересными для **Map** являются два метода:

1. **get** — получить объект по ключу
2. **put** — поместить объект с ключом в коллекцию. Если такой ключ уже есть, то старый объект будет заменен на новый

Как и любая коллекция, **Map** тоже позволяет пройтись по списку своиъ элементов — причем как по ключам, так и величинам. И даже по парам/двойкам — ключ/значение. Для такого доступа соответсвенно используются методы **keySet(), values(), entrySet()**.
Давайте не будем растекаться мыслью по древу и посмотрим несложный пример, который демонстрирует основные операции при работе с **Map**.
В нашем примере мы создадим коллекцию из объектов типа **WebSiteUser**, у которого ключом будет поле e-mail.
Сам класс **WebSiteUser** представляет собой достаточно простую конструкцию из Имени, фамилии, адреса электронной почты и телефона. Что-то вроде такого:

```
1  package edu.javacourse.collection;
2
3  public class WebSiteUser
4  {
5      private String email;
6      private String firstName;
7      private String lastName;
8      private String phone;
9
10     public WebSiteUser(String email, String firstName, String lastName, String phone) {
11         this.email = email;
12         this.firstName = firstName;
13         this.lastName = lastName;
14         this.phone = phone;
15     }
16
17     public String getEmail() {
18         return email;
19     }
20
21     public String getFirstName() {
22         return firstName;
```

```
    }

    public String getLastName() {
        return lastName;
    }

    public String getPhone() {
        return phone;
    }
}
```

```
package edu.javacourse.collection;

import java.util.HashMap;
import java.util.Map;

public class MapExample
{
    public static void main(String[] args) {
        // HashMap - наверно самый распространенный класс - но есть и другие.
        Map<String, WebSiteUser> mapItems = new HashMap<>();

        // Создаем объекты
        WebSiteUser w1 = new WebSiteUser("andy@google.com", "Andy", "Collah", "+1-467-98763245");
        WebSiteUser w2 = new WebSiteUser("john@yahoo.com", "John", "Smith", "+1-467-9870574426");
        WebSiteUser w3 = new WebSiteUser("cris@gmail.com", "Cris", "Balen", "+1-632-4267473426");

        // Помещаем объекты в коллекцию - ключом является поле email
        mapItems.put(w1.getEmail(), w1);
        mapItems.put(w2.getEmail(), w2);
        mapItems.put(w3.getEmail(), w3);
        // Еще раз вставляем объект, но с другим ключом
        mapItems.put("other@yandex.ru", w3);

        // Получить объет по ключу
        System.out.println("Получить пользователя по ключу");
        WebSiteUser user = mapItems.get("cris@gmail.com");
        System.out.println("User:" + user.getFirstName() + ":" + user.getLastName());

        // Пройти по коллекции из ключей - доступ к коллекции ключей через keySet()
        System.out.println();
        System.out.println("Список пользователей по ключу:");
        for(String email : mapItems.keySet()) {
            // Получить объект по ключу
```

```
            WebSiteUser u = mapItems.get(email);
            System.out.println("User:" + email + ", " + u.getFirstName() + ":" + u.getLastName());
        }

        // Пройти по коллекции из значений - доступ к коллекции значений через values()
        System.out.println();
        System.out.println("Список пользователей по значению:");
        for(WebSiteUser us : mapItems.values()) {
            System.out.println("User:" + us.getFirstName() + ":" + us.getLastName());
        }

        // Пройти по коллекции из пар - доступ к коллекции через entrySet()
        System.out.println();
        System.out.println("Список пользователей в виде пар:");
        for(Map.Entry<String, WebSiteUser> us : mapItems.entrySet()) {
            System.out.println("User:" + us.getKey() + ", " + us.getValue().getFirstName() + ":" + us.getValue().getLas
        }
    }
}
```

Суть примера — показать, как пользоваться основными методами интерфейса **Map**. В примере мы использовали класс **java.util.HashMap**. Существуют еще несколько реализаций — например **TreeMap, Hashtable, ConcurrentHashMap**. Думаю, что вы можете самостоятельно помсотреть их опимсание в документации и понять их особенности, которые пригодяться вам в той или иной ситуации.

Класс Properties

Класс **Properties** — одна из реализаций интерфейса **Map**, которая имеет ряд особенностей и выполняет достаточно интересную роль — с его помощью можно легко создавать конфигурационные файлы и к тому же создавать программы, которые поддерживают многоязычность. Сначала посмотрим вариант с конфигурационным файлом. Для этого класса конфигуарционный файл — это обычный текстовый файл, в котором свойства хранятся в виде ключ/значение через знак «=». Вот так выглядит такой файл:

```
# Comments - file "simple.properties"
up.button.title=UP
dn.button.title=DOWN
```

Рассмотрим небольшой пример, который продемонстрирует нам как работать с конфигурационным файлом, а потом мы увидим как можно сделать наше приложение многоязычным. Пример конечно же не может претендовать на роль идеального, но думаю, что он донесет основные принципы — а это для меня главное.
Для начала мы создадим графическое приложение, у которого будет форма в двумя кнопками. Наша задача — сделать текст на этих кнопках

настраиваемым через конфигурационный файл. Если бы мы создавали наше приложение без возможности конфигурации, то оно могло бы выглядеть как-то так:

```
package edu.javacourse.collection;

import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class PropertiesExample extends JFrame
{
    public PropertiesExample() {
        JButton up = new JButton("UP");
        JButton down = new JButton("DOWN");

        add(up, BorderLayout.NORTH);
        add(down, BorderLayout.SOUTH);

        setBounds(200, 200, 500, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        PropertiesExample pe = new PropertiesExample();
    }
}
```

Как видите, текст на кнопках жестко «прошит» в коде. Давайт воспользуемся нашим конфигурационным файлом, который поместим в корневую папку с нашим классом. Если вы создаете проект в NetBeans или в иной IDE, то этот файл должен лежать непосредственно в папке с проектом. Вот наш конфигурационный файл с именем **simple.properties**

```
up.button.title=UP button
dn.button.title=DOWN button
```

Модифицированный пример с комментариями. Работу с файлами мы пока не проходили, но примите эту конструкцию пока на веру. Когда мы будем проходить файловые операции, мы раскроем, что и как.

```
```

```java
package edu.javacourse.collection;

import java.awt.BorderLayout;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;
import javax.swing.JButton;
import javax.swing.JFrame;

public class PropertiesExample extends JFrame
{
    public PropertiesExample() {
        try {
            // Создаем объект
            Properties pr = new Properties();
            // Загружаем данные из файла - позже узнаем более подробно про файлы
            pr.load(new FileReader("simple.properties"));

            // Получаем свойства по именам - это же по сути Map
            String upText = pr.getProperty("up.button.title");
            String dnText = pr.getProperty("dn.button.title");

            // Создаем кнопки с указанными названиями
            JButton up = new JButton(upText);
            JButton dn = new JButton(dnText);
            add(up, BorderLayout.NORTH);
            add(dn, BorderLayout.SOUTH);
        } catch (IOException io_ex) {
            io_ex.printStackTrace(System.out);
        }

        setBounds(200, 200, 500, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        PropertiesExample pe = new PropertiesExample();
    }
}
```

Как видите никаких сложностей нет — просто загрузили файл и он сам разобрался на составные части. Удобно.

Доступ к конфигурации через ресурсы

У меня были сомнения по поводу этого раздела — мы еще не так много прошли, но я подумал, что будет не страшно, если мы какие-то моменты пока просто примем как есть (та же загрузка из файла). Давайте поступим также и в случае с понятием «ресурс». Если не отвлекаться далеко и глубоко, то ресурсом является какая-либо информация в файле (текстовом, графическом или ином), который составляет с нашими классами одним целым. Т.е. есть классы и вместе с ними есть ресурсы, которые этим классам нужны. Ключевым моментом в этой ситуации являетя универсальность доступа — т.к. файлы/ресурсы и классы как бы одно целое, то они могут перемещаться вместе и что самое интересное — путь к этим ресурсам будет всегда начинаться от корня расположения классов. Т.е. положили мы наши классы и конфигурационный файл в папку **Lesson** и получили вот такое дерево:

```
- Lesson
    - edu
        - javacourse
            - collection
                PropertiesExample.class
                simple.properties
```

Если вы (как и я в даном курсе) разрабатываете примеры в NetBeans, то файл **simple.properties** должен лежать в той же папке, что и файл **PropertiesExample.java** — **src/edu/javacourse/collection**. При сборке он будет помещен вместе с классами. Теперь мы можем обращаться уже к РЕСУРСУ **simple.properties** иначе. Смотрим код:

```
package edu.javacourse.collection;

import java.awt.BorderLayout;
import java.util.PropertyResourceBundle;
import javax.swing.JButton;
import javax.swing.JFrame;

public class PropertiesExample extends JFrame
{
    public PropertiesExample() {
        // Загружаем данные из ресурса - обратите внимание на путь до ресурса
        // И пока можете принять такую форму загрузки как данность
        PropertyResourceBundle pr = (PropertyResourceBundle)
                PropertyResourceBundle.getBundle("edu.javacourse.collection.simple");

        // Получаем свойства по именам - это же по сути Map
        String upText = pr.getString("up.button.title");
        String dnText = pr.getString("dn.button.title");

        // Создаем кнопки с указанными названиями
        JButton up = new JButton(upText);
        JButton dn = new JButton(dnText);
        add(up, BorderLayout.NORTH);
```

```
        add(dn, BorderLayout.SOUTH);

        setBounds(200, 200, 500, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        PropertiesExample pe = new PropertiesExample();
    }
}
```

Обратите внимание на то, что к ресурсу путь указывается через точку и мы не указываем расширение файла «properties».
В NetBeans вы можете заглянуть в папку **build/classes** — там вы увидите как раз ту структуру, которую я уже показывал

```
- build
    - classes
        - edu
            - javacourse
                - collection
                    PropertiesExample.class
                    simple.properties
```

В принципе пока достаточно будет осознать, что в случае, когда вам в приложени требуется какой-то ресурс — картинка, текст, свойства — то это отличный кандидат на «ресурс». Такой файл надо размещать возле классов и обращаться к нему через **ResourceBundle**.

Многоязычные ресурсы

Допускаю, что данный раздел несколько опережает события, но раз мы уж едобрались до понятия «ресурс», то приведу пример и многоязычности. Возможно, что мы еще вернемся к этому разделу, но раз уж так сложилось — не буду отказываться. Если что — напишите, что вам это не понравилось. Я постараюсь это учесть.
Итак, Java предоставляет механизм многоязычных свойств. Делается это достаточно просто. Существует специальный класс **java.util.Locale**. Этот класс содержит настройки, которые характеризуют тот или иной язык (и даже культуру) — например, написание даты в США и в России сильно отличаются — в США месяц/день/год, в России — день.месяц.год. Существует даже термин «локаль», который говорит, под каким «языком» вы работаете — русская локаль, английская локаль.
Каждая локаль имеет краткое название. Например английская — «us». Русская — «ru». Но т.к. например, английская локаль имеет распространение в нескольких странах, то существует еще более «точная настройка». Например США — en_US, Канада — en_CA, Великобритания — en_GB. Есть даже третий уровень, но он встречается уже редко.
Так вот — когда вы загружаете ресурс, то он загружается согласно определенным правилам.

1. Определяется текущая локаль — по умлочаанию ваша операционная система всегда имеет локаль. Вот такой вызов покажет вам, кака я локаль используется по умолчанию

```
System.out.println(Locale.getDefault().toString());
```

2. К имени ресурса прибавляется локаль и сначала ищется файл с таким именем. Например локаль по умолчанию — en_US (для анлийского языка и США). В этом случае на самом деле сначала мы будем искать файл **simple_en_us.properties**. Если не найдем — будем искать **simple_en.properties**. И только потом **simple.properties**

.

Давайте создадим три конфигурационных файла:
simple.properties

```
up.button.title=UP button !!!
dn.button.title=DOWN button !!!
```

simple_en.properties

```
up.button.title=UP button !!! English
dn.button.title=DOWN button !!! English
```

simple_en_us.properties

```
up.button.title=UP button !!! English and USA
dn.button.title=DOWN button !!! English and USA
```

Ну и сам класс, который загружает ресурс с указанием локали

```
package edu.javacourse.collection;

import java.awt.BorderLayout;
```

```java
import java.util.Locale;
import java.util.PropertyResourceBundle;
import javax.swing.JButton;
import javax.swing.JFrame;

public class PropertiesExample extends JFrame
{
    public PropertiesExample() {
        // Загружаем данные из ресурса с указанием локали
        PropertyResourceBundle pr = (PropertyResourceBundle)
                PropertyResourceBundle.getBundle("edu.javacourse.collection.simple", new Locale("en_US"));

        // Получаем свойства по именам - это же по сути Map
        String upText = pr.getString("up.button.title");
        String dnText = pr.getString("dn.button.title");

        // Создаем кнопки с указанными названиями
        JButton up = new JButton(upText);
        JButton dn = new JButton(dnText);
        add(up, BorderLayout.NORTH);
        add(dn, BorderLayout.SOUTH);

        setBounds(200, 200, 500, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        PropertiesExample pe = new PropertiesExample();
    }
}
```

В качестве самостоятельной работы попробуйте изменять локаль и удалять файлы.
ВАЖНО !!! При работе с локализацией есть один момент — если вы собираетесь использовать русские буквы, то вы не сможете писать из напрямую. Вам необходимо использовать никоды для русских букв. Например, если в русской локали заголовоки кнопок хотелось бы сделать такими:

```
up.button.title=Верхняя кнопка
dn.button.title=Нижняя кнопка
```

то в реальности файл будет выглядеть вот так:

```
up.button.title=\u0412\u0435\u0440\u0445\u043d\u044f\u044f \u043a\u043d\u043e\u043f\u043a\u0430
dn.button.title=\u041d\u0438\u0436\u043d\u044f\u044f \u043a\u043d\u043e\u043f\u043a\u0430
```

Не очень красиво и понятно, но не волнуйтесь. Во-первых тот же NetBeans отслеживает такую ситуацию автоматически. IDEA тоже — правда ей надо об этом сказать в настройках. Во-вторых — в составе JDK существует даже специальная утилита **native2ascii**, которая преобразует файл с русскими буквами в такой вот цифровой код.

Домашнее задание

Сделайте русскую локализацию нашего примера с кнопками. Удачи.

И теперь нас ждет следующая статья: Список контактов — GUI приложение.

7 comments to *Коллекции — продолжение*

- 
  Январь 27, 2018 at 00:45
  Максим says:

  А где описание класса WebSiteUser?

  Reply
  - 
    Январь 29, 2018 at 09:33
    admin says:

    Я думал, что из контекста понятно, как должен выглядеть этот класс. Но видимо есть смысл добавить — добавил. Спасибо за замечание.

    Reply
    - 
      Январь 30, 2018 at 11:29

Максим says:

Спасибо. Обычно у Вас все так подробно расписано, что отсутствие описания несколько удивило. А оказывается это был элемент самостоятельной работы)

Reply

- 
  Март 13, 2018 at 06:30
  MadFisher says:

  В первом примере в конструкторе для MyClass2 скорее всего опечатка «MyClass1»

  Reply

  - 
    Март 13, 2018 at 12:11
    admin says:

    Спасибо, исправил.

    Reply

- 
  Сентябрь 5, 2018 at 22:08
  andy says:

  В блоке:

  System.out.println(«Печать через итератор списка от конца к началу»);
  for(ListIterator li = test.listIterator(test.size()); li.hasPrevious();) {
  String s = li.previous();
  System.out.println(s);
  }
  System.out.println(«Печать через итератор списка от начала к концу»);

```
for( ListIterator li = test.listIterator(); li.hasNext(); ) {
String s = li.next();
System.out.println(s);
}
```

В одном случае получаем лист-итератор как .listIterator(test.size()), а в другом как .listIterator(); — тут все корректно?

Reply

- 

  Сентябрь 6, 2018 at 15:54
  admin says:

  Вы уж как-нибудь повнимательнее и пытайтесь понимать и в какой-то степени отгадывать.
  В первом случае мы устанавливаем итератор на конец — по сути на какую-то точку в списке — и идем к началу. Значит указываем эту точку.
  Во втором случае становимся в начало — здесь не требуется ничего указывать.

  Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

два × = 10

Add comment