12. 泛型

12.1. 为什么需要泛型?

- 为了使集合能够记住集合内各元素的类型，且能够达到只要编译时不出现问题，运行就不会出现java.lang.ClassCastException异常——使用泛型

12.2. 什么是泛型

- JDK1.5之后出现了新的技术——泛型（Generic），此技术的最大特点是类中的类型可以由外部决定。泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传实参。那么参数化是指将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）

12.3. 自定义泛型接口、泛型类

- 泛型类和方法的定义：

```
// 泛型类的定义
Class Node<T> {
    private T data;

    public Node() {}

    public Node(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}

// 泛型接口的定义
```

```
interface Shopping<T> {
    public void shopping(T name);
}
```

12.4. 通配符

```
// 问题
Node<Number> c1 = new Node<Number>();
Node<Integer> c2 = new Node<Integer>();
c1 == c2;// 此时无法转换

public static void getData(Node<Number> data) {
    System.out.ptintln("data=" + data.getData());
}
```

- 此时可以使用通配符：“？”表示的是可以接收任意的泛型类型，但是只能接收输出，并不能修改。

```
public static void getData(Node<?> data) {
    System.out.println("data=" + data.getData());
}
// 使用通配符可以引用其他各种参数化类型，通配符定义的变量主要用作引用，可以调用与参数无关的方法，不能调用与参数有关的方法
```

- 泛型上限就指一个操作泛型最大的操作父类。泛型的上限通过以下语法完成：

```
// ? extends 类
public static void getUpperNumberData(Node<? extends Number> data) {
    // 只能是Number类及其子类
    System.out.println("data=" + data.getData());
}
```

- 泛型的下限指的是只能设置其具体的类或父类。设置语法如下：

```
// ? super 类
```

12.5. 泛型方法

- 泛型除了在类中定义之外，还可以在方法上定义，而且在方法上使用泛型，此方法所在的类不一定是泛型的操作类

```
// 定义一个方法，实现任意类型数组中两个位置值得调换
public static <T> T[] function(T[] array, int i, int j) {
    T temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    return array;
}
```

12.6. 泛型的嵌套使用

- 在使用集合时，我们可以使用泛型嵌套：

```
Set<Entry<Integer, String>> entrys = map.entrySet();
```

12.7. 示例

```
package day12_泛型;

import org.junit.Test;

import java.util.*;

/**
 * @author xiao儿
 * @date 2019/9/3 19:16
 * @Description GenericDemo
 */
public class GenericDemo {
    @Test
    public void test1() {
        List<String> list = new ArrayList<>();
        list.add("Tom");
        // list.add(10);
        // list.add(new Object());

        for (int i = 0; i <list.size(); i++) {
            // 如果我们不确定集合中的元素类型，那么我们需要在处理元素时进行判断元素的类型，才可以执行不同的操作
        }
    }

    @Test
    public void test2() {
        Node<Number> numberNode = new Node<>();
        Node<Integer> integerNode = new Node<>();
    }

    @Test
    public void test3() {
        Node<Number> numberNode = new Node<>(10);
        Node<Integer> integerNode = new Node<>(20);

        getData(numberNode);
        // 不支持赋值
        // getData(integerNode);
        // numberNode = integerNode;
        getDate2(integerNode);
        getUpperNumberData(numberNode);
        getUpperNumberData(integerNode);
    }

    public static void getData(Node<Number> node) {
        System.out.println(node.getData());
    }

    /**
```

```java
     * 使用通配符定义泛型类型，此时只能输出，不能修改
     * @param node
     */
    public static void getDate2(Node<?> node) {
        // node.setData(20);
        System.out.println(node.getData());
    }

    public static void getUpperNumberData(Node<? extends Number> data) {
        // 只能是 Number 类或其子类
        System.out.println("data=" + data.getData());
        // data.setData(10);
    }

    /**
     * 泛型方法
     * @param array
     * @param i
     * @param j
     * @param <T>
     * @return
     */
    public static <T> T[] function(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        return array;
    }

    @Test
    public void test4() {
        String[] arrays = {"Tom", "Lily", "Bin", "Jack"};
        String[] strs = function(arrays, 0, 2);
        System.out.println(Arrays.toString(strs));
    }

    @Test
    public void test5() {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Tom");
        map.put(2, "Lily");

        Set<Map.Entry<Integer, String>> entrySet = map.entrySet();
        for (Map.Entry entry : entrySet) {
            System.out.println(entry.getKey() + "-" + entry.getValue());
        }
    }
}
```