讲义09：泛型与容器类

主要内容

- 讲授内容
  - 泛型概念与类型参数
  - 泛型类、泛型方法和泛型接口
  - 泛型限制和泛型通配符
  - Java容器基本概念
  - 列表接口List及实现类：ArrayList、LinkedList
  - 集合接口Set及实现类：HashSet、TreeSet
  - 映射接口Map及实现类：HashMap、TreeMap
- 内容资料来源
  - 教材第12章
  - Java API文档

1 泛型

- 泛型(genericity)是指数据的类型参数化
- 通过为类、接口和方法设置类型参数实现泛型
- 泛型使类或方法可以对不同类型的对象进行操作

- JDK 5开始支持泛型

1.1 泛型的概念

一个类或方法可以处理多种类型对象可以使用类型转换的方式。

```
public int compare(Object p1, Object p2) { ... }
```

上面定义的方法compare的2个参数可以接受任何类型的对象。

缺点：使用类型转换方式时，程序中使用对象实例时，需要进行向下强制类型转换，因此需要编写代码在程序运行时进行类型判断，来确保类型转换成功，不发生ClassCastException。

Java从JDK 5开始支持泛型，允许把数据类型作为参数，为泛型类和泛型方法指定可以处理的对象类型。

优点：在编译期检查类型是否错误，而不是在运行期检查。

1.1 泛型的概念

定义泛型的语法(头部定义)

泛型类：　[修饰符] class 类名<T>

泛型接口：[修饰符] interface 接口名<T>

泛型方法：[修饰符] [static] <T> 返回类型 方法名(T 参数)

T是泛型中的“类型参数”，在类、接口或方法中定义类型参数T后，就可以在相应的各个语句部分中使用参数T。

1.2 泛型类定义及应用

示例：使用泛型定义一个基于数组的栈

```
public class ArrayStack<E> {
    // 存放栈元素的数组，默认是Object类型
    private Object[] elements = {};
    // 入栈方法 ----------------------------------------------------
    public void push(E element) {
        // 检查栈是否已经满了
        if (top == elements.length) {
            Object[] temp = new Object[elements.length + 16];
            System.arraycopy(elements, 0, temp, 0, elements.length);
            elements = temp;
        }
        elements[top++] = element; // 入栈
    }
    // 栈顶元素方法 ------------------------------------------------
    public E peek() {
        if (top == 0) {
            throw new EmptyStackException(); // 如果栈为空抛出异常
        }
        return (E) elements[top];  // 注意要强制类型转换
    }
    // 省略部分代码
}
```

完整代码：examples/lecture09/genericity_01

1.2 泛型类定义及应用

- 创建泛型对象时不传递类型，默认是Object

```
ArrayStack stackObject = new ArrayStack();
```

- 创建泛型对象时传递类型，栈内元素中指定类及其子类类型

```
ArrayStack<String> stack1 = new ArrayStack<String>();
ArrayStack<Number> stack2 = new ArrayStack<Number>();
```

- 从JDK7开始可以省略构造方法后面<>中的类

```
ArrayStack<String> stack1 = new ArrayStack<>();
ArrayStack<Number> stack2 = new ArrayStack<>();
```

1.3 泛型方法的定义及应用

示例：使用泛型定义一个对数组进行升序冒泡排序的静态方法

```
public static <E> void bubbleSort(E[] elements) {
    for (int i = 0; i < elements.length - 1; i++) {
        for (int j = 0; j < elements.length - 1 - i; j++) {
            if (((Comparable<E>) elements[j]).compareTo(elements[j + 1]) > 0) {
                E t = elements[j];
                elements[j] = elements[j + 1];
                elements[j + 1] = t;
            }
        }
    }
}
```

完整代码：examples/lecture09/genericity_02

1.3 泛型方法的定义及应用

```
Integer[] arr01 = { 33, 2, 18, 66, 8, 7 };
String[] arr02 = {"Java", "Python", "C++", "Ruby"};
```

如下是调用泛型静态方法

```
Demo.<Integer>bubbleSort(arr01);
Demo.<String>bubbleSort(arr02);
```

可以使用如下形式省略调用时的泛型指示

```
Demo.bubbleSort(arr01);
Demo.bubbleSort(arr02);
```

1.4 限制泛型的可用类型

定义泛型时，默认情况下任何类型都可以传递给泛型参数，如果要限制传入的类型，可以使用以下语法：

<T extends 类>　　传入类型是指定类或其子类

<T extends 接口>　传入类型是指定实现了指定接口的类

说明：无论是类还是接口都使用 extends

例如：

```
public class ArrayStack<E extends Number>
public static <E extends Comparable<E>> void bubbleSort(E[] elements)
```

1.5 泛型通配符

为什么使用通配符?

Java中如果Dog extends Animal，则：

- 对象，即 Animal a = new Dog(); 正确
- 数组，即 Animal[] a = new Dog[10]; 正确
- 泛型，即 Demo<Animal> a = new Demo<Dog>(); 错误

第3种情况下，可以使用通配符解决。

通配符用于泛型类声明变量(方法形参数)，主要有3种用法：

- 无边界的通配符：　　泛型类名<?> var;
- 固定上边界的通配符：泛型类名<? extends E> var;
- 固定下边界的通配符：泛型类名<? super E> var;

说明：E可以是任何类或接口

1.5 泛型通配符

示例背景介绍

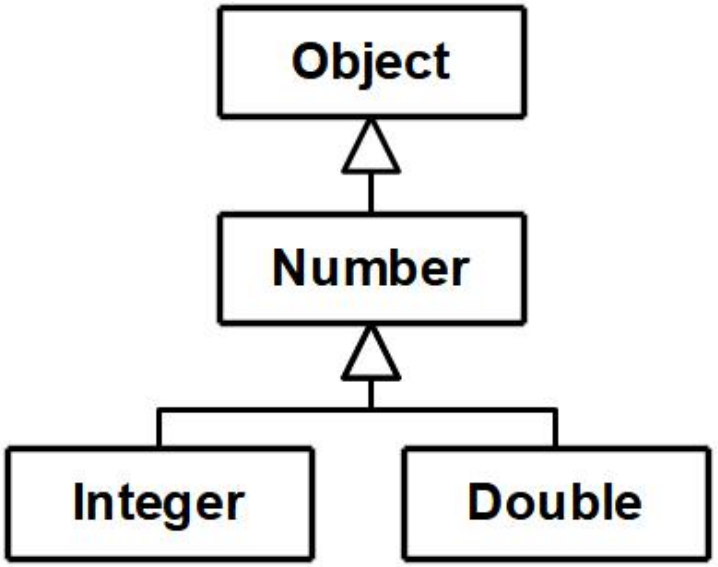
泛型类的定义

```
public class Demo<T> {
    private T data;

    public T getData() {
        return data;
    }

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

作为类型参数的类



完整代码：
examples/lecture09/genericity_03

1.5 泛型通配符

考虑一个使用泛型类Demo声明的变量(形参)

```
public static void output(Demo<___> d) {
    System.out.println(d.getData());
}
```

如果声明d时不使用通配符，则只能接收特定类型参数的泛型对象，例如：

Demo<Object> d 只能接收 Demo<Object>类型的对象
Demo<Integer> d 只能接收 Demo<Integer>类型的对象

1.5.1 无边界的通配符

```
public static void output(Demo<?> d) {
    System.out.println(d.getData());
}
```

形参d接收任何类型参数的泛型对象，例如：

```
Demo<Object> dObject = new Demo<Object>();
Demo<Number> dNumber = new Demo<Number>();
Demo<Integer> dInteger = new Demo<Integer>();
Demo<Double> dDouble = new Demo<Double>();
Demo<String> dString = new Demo<String>();
output(dObject);   //正确
output(dNumber);   //正确
output(dInteger);  //正确
output(dDouble);   //正确
output(dString);   //正确
```

1.5.2 固定上边界的通配符

```
public static void output(Demo<? extends Number> d) {
    System.out.println(d.getData());
}
```

形参d接收类型参数为Number或其子类的泛型对象，例如：

```
Demo<Object> dObject = new Demo<Object>();
Demo<Number> dNumber = new Demo<Number>();
Demo<Integer> dInteger = new Demo<Integer>();
Demo<Double> dDouble = new Demo<Double>();
Demo<String> dString = new Demo<String>();
output(dObject);   //错误
output(dNumber);   //正确
output(dInteger);  //正确
output(dDouble);   //正确
output(dString);   //错误
```

1.5.3 固定下边界的通配符

```
public static void output(Demo<? super Number> d) {
    System.out.println(d.getData());
}
```

形参d接收类型参数为Number到Object的泛型对象，例如：

```
Demo<Object> dObject = new Demo<Object>();
Demo<Number> dNumber = new Demo<Number>();
Demo<Integer> dInteger = new Demo<Integer>();
Demo<Double> dDouble = new Demo<Double>();
Demo<String> dString = new Demo<String>();
output(dObject);   //正确
output(dNumber);   //正确
output(dInteger);  //错误
output(dDouble);   //错误
output(dString);   //错误
```

1.6 继承泛型类与实现泛型接口

Java中，定义类时可以继承一个父类或实现多个接口。
其中，父类可以是泛型类，接口可以是泛型接口。

以下面的例子来说明具体的使用方法：

```
public abstract class GenericClass<T1,T2> {
    T1 age;
    public abstract void test(T2 name);
}
```

```
public interface GenericInterface<T> {
    public abstract void m(T para);
}
```

1.6.1 继承泛型类

1.泛型全部保留，子类为泛型子类

```
class F1<T1, T2> extends GenericClass<T1,T2>{
    @Override public void test(T2 name) { }
}
```

2.泛型部分保留，部分确定类型，子类为泛型子类

```
class F2<T2> extends GenericClass<Integer,T2>{
    @Override public void test(T2 name) { }
}
```

1.6.1 继承泛型类

3.不保留，全部确定具体类型，子类是非泛型类

```
class F3 extends GenericClass<Integer,String>{
    @Override  public void test(String name) {  }
}
```

4.没有类型(擦除,类似于Object)，子类是非泛型类

```
class F4 extends GenericClass{
    @Override  public void test(Object name) {  }
}
```

5.子类可以增加自己的泛型

```
class F5<T1, T2, T3> extends GenericClass<T1,T2>{
    T3 data;
    @Override public void test(T2 name) {    }
}
```

1.6.2 实现泛型接口

1.全部或部分保留接口泛型，子类为泛型子类

```
class F1<T> implements GenericInterface<T>{
    @Override public void test(T para) { }
}
```

2.全部确定接口泛型的具体类型，子类是非泛型类

```
class F2 implements GenericInterface<String>{
    @Override  public void test(String para) { }
}
```

1.7 泛型使用的注意事项

1. 不能使用泛型的类型参数创建对象

```
public class Demo<T> {
  T obj = new T();  //错误
}
```

2. 不能使用泛型的类型参数创建数组对象

```
public class Demo<T> {
  T[] arr = new T[个数];  //错误
}
```

3. 不能在静态(变量、方法、初始化)使用泛型的类型参数

```
public class Demo<T> {
  static T data; //错误
}
```

4. 定义异常类不能使用泛型

```
public class MyEx<T> extends Exception {} //错误
```

2 Java的容器类

Java语言通过集合框架(Collections Framework)提供常用的数据结构。

下面列出了部分集合接口与实现了接口的类(java.util包):

接口	哈希	可变数组	平衡树	链表	哈希+链表
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

2.1 java.util.ArrayList<E>

一个基于可变长数组的线性表，常用构造方法：

- public ArrayList()
  构造一个初始容量为 10 的空列表
- public ArrayList(int initialCapacity)
  构造一个初始容量为 initialCapacity 的空列表

声明ArrayList时使用ArrayList<ClassName>，表示存储元素的类是ClassName或它的子类。

例如：

```
ArrayList<Circle> list01 = new ArrayList<>();
ArrayList<Circle> list02 = new ArrayList<>(32);
```

2.1 java.util.ArrayList<E> - 基本操作方法

- public boolean add(E e)
  在末尾追加一个元素 e, 返回 true
- public void add(int index, E e)
  在下标 index 处插入一个元素 e
- public E remove(int index)
  删除并返回下标 index 处的元素
- public boolean remove(Object o)
  删除等于 o 的第1个元素(使用 equals 判断), 返回值表示是否成功
- public E get(int index)
  返回下标 index 处的元素
- public E set(int index, E e)
  使用 e 替换 下标 index 处的元素
- public void clear()
  清空整个线性表

2.1 java.util.ArrayList<E> - 查询操作

- public boolean contains(Object o)
  判断线性表中是否包含 o , 使用equals判断
- public int indexOf(Object o)
  返回第1个等于 o 的元素的下标, -1表示查找失败
- public int lastIndexOf(Object o)
  返回最后1个等于 o 的元素的下标, -1表示查找失败

- public boolean isEmpty()
  判断线性表中元素个数是否为0
- public int size()
  返回线性表中元素的个数

2.1 java.util.ArrayList<E> - 示例

```
ArrayList<Person> list = new ArrayList<>();
Student s = new Student("张三", "华南农业大学");
Teacher t = new Teacher("李四", "程序设计");

list.add(s);
list.add(t);

for(Person p : list) [
    System.out.println(p.sayHi());
]

Student stud = new Student("张三", "华南理工大学");
System.out.println(list.contains(stud));

System.out.println(list.size());
```

示例代码：examples\Lecture09\collection_demo

2.2 java.util.HashSet<E>

一个基于Hash Table实现的set, 不允许重复值, 常用构造方法:

- public HashSet()
  构造一个初始容量为 16，扩容因子为0.75 的空set
- public HashSet(int initialCapacity)
  构造一个初始容量为 initialCapacity, 扩容因子为0.75的空set
- public HashSet(int initialCapacity, float loadFactor)
  构造一个初始容量为 initialCapacity, 扩容因子为loadFactor 的空set

例如:

```
HashSet<Circle> set01 = new HashSet<>();
HashSet<Circle> set02 = new HashSet<>(64);
```

2.2 java.util.HashSet<E> - 基本操作

- public boolean add(E e)
  增加 set 中没有的元素 e, 返回 true; 如果e已经存在, 返回 false
- public boolean remove(Object o)
  删除等于 o 的元素(使用 equals 判断), 返回值表示是否成功
- public void clear()
  清空整个 set
- public boolean contains(Object o)
  判断 set 中是否包含 o , 使用equals判断
- public boolean isEmpty()
  判断 set 中元素个数是否为0
- public int size()
  返回 set 中元素的个数

课后工作

- 结合教材和Java API文档，要求掌握：
  - 泛型的定义与使用
  - 容器类的概念与常用的线性表、集合、映射的使用
- 自行练习教材的上机实践和习题