# 第七章: 集合与策略、迭代器模式

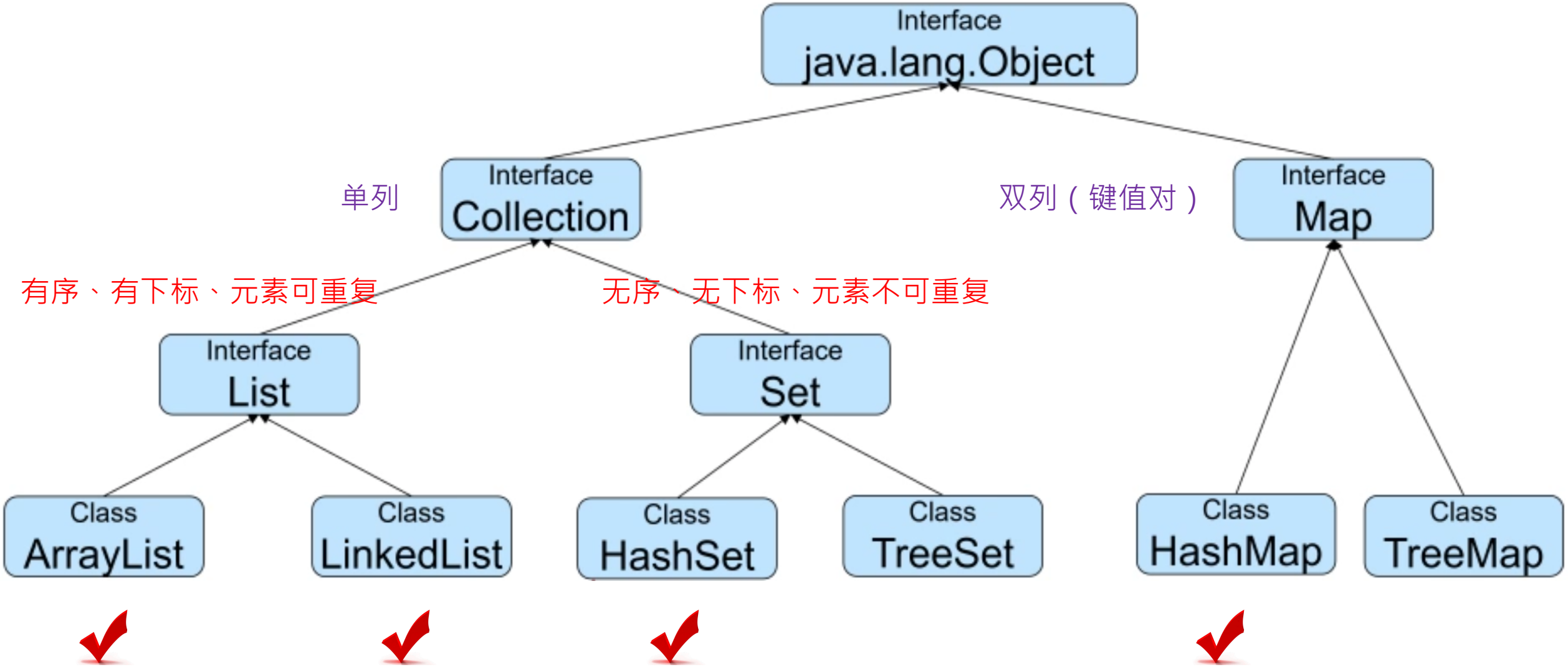
# 第七章：集合与策略、迭代器模式

- 集合类概述
- List接口及其标准实现
- 类ArrayList与LinkedList
- Set与Map接口
- 策略模式
- 迭代器模式

# 集合概述

- Java语言的java.util包中提供了一些集合类，这些集合类又被称为**容器**。
- 集合是**对象**的容器，定义了对多个对象进行**操作**的常用方法。
- **与数组的区别：**
  - **数组**的长度是**固定**的，**集合**的长度是**可变**的；
  - **数组**用来存放**基本类型**的数据和**对象**的引用，**集合**只能用来存放**对象**的引用;
- 通过java.util包提供
  - 比如import java.util.ArrayList; //引入ArrayList类

# 什么是集合


Interface
java.lang.Object
单列
Interface
Collection
双列（键值对）
Interface
Map
有序、有下标、元素可重复
无序、无下标、元素不可重复
Interface
List
Interface
Set
Class
ArrayList
Class
LinkedList
Class
HashSet
Class
TreeSet
Class
HashMap
Class
TreeMap

# List接口

- List接口定义了一个有序的对象集合，特点：有序、有下标、元素可以重复。
- ArrayList 类是一个可以动态修改的数组，与普通数组的区别就是它是没有固定大小的限制，我们可以添加或删除元素。

```
import java.util.ArrayList; // 引入 ArrayList 类

ArrayList<E> objectName =new ArrayList<E>();     // 初始化
```

E: 泛型数据类型，用于设置 objectName 的数据类型

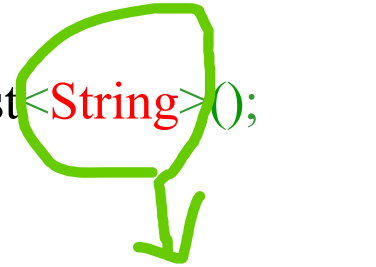
- ArrayList 是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

# ArrayList类

实例：

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites);
    }
}
```



可省略

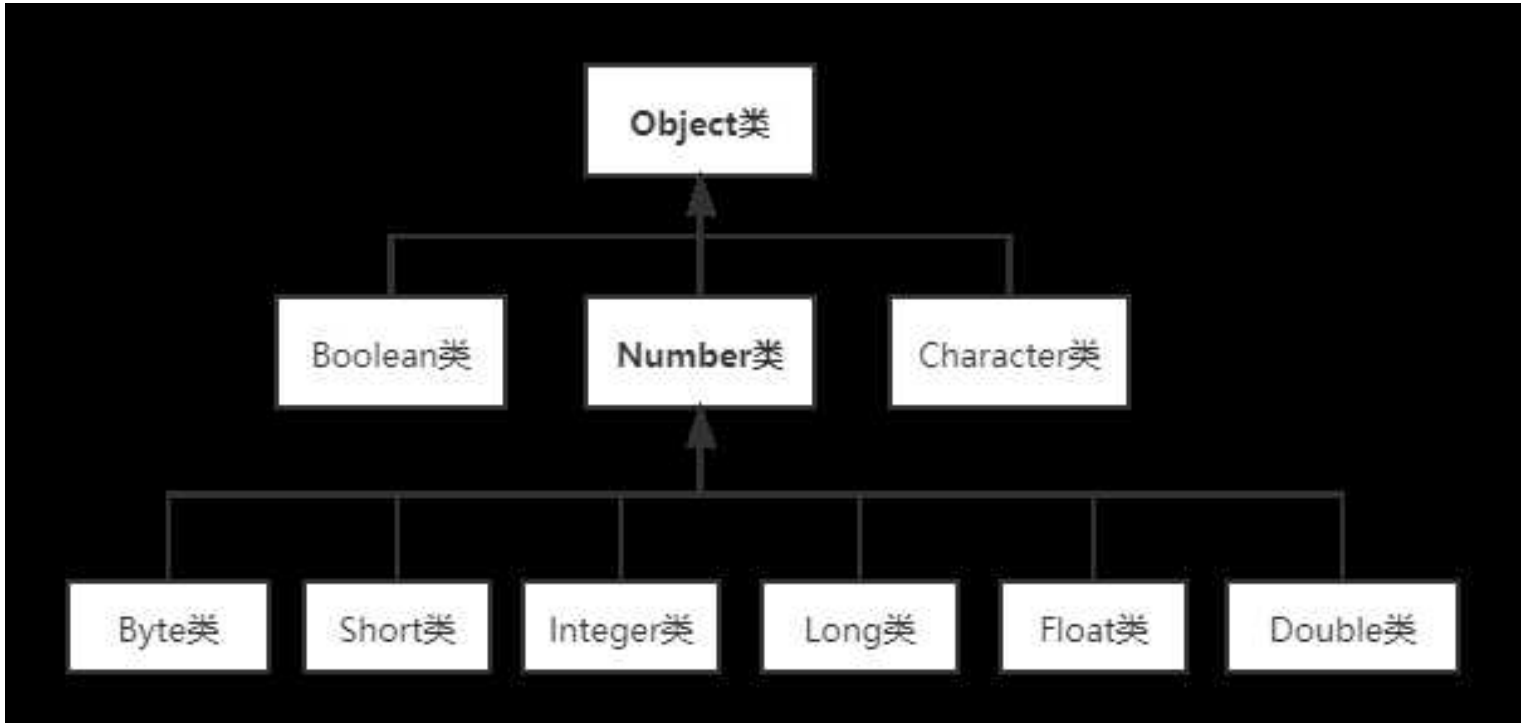
输出为：[Google, Amazon, Taobao, Weibo]

# 互动小问题

如果只想放**整数**，下面代码怎么修改？

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<Integer? int?> sites = new ArrayList<>();
        sites.add(1);
        sites.add(23);
        sites.add(456);
        System.out.println(sites);
    }
}
```



# 互动小问题

数组能不能存放**不同类型**的数据？

# ArrayList类

在列表中间增加元素：

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add(“Google”);
        sites.add(“Amazon”);
        sites.add(“Taobao”);
        sites.add(1, "Weibo");
        System.out.println(sites);
    }
}
```

输出为： [Google, Weibo, Amazon, Taobao]

# ArrayList类

**删除元素**：如果要删除 ArrayList 中的元素可以使用 remove() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        sites.remove(3); // 删除第四个元素
        sites.remove("Google"); //直接删除
        System.out.println(sites);
    }
}
```

输出为： [Amazon, Taobao]

# ArrayList类

**修改元素**：访问 ArrayList 中的元素可以使用 set() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        sites.set(2, "Wiki"); // 第一个参数为索引位置，第二个为要修改的值
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Wiki, Weibo]

# ArrayList类

**访问元素**：访问 ArrayList 中的元素可以使用 get() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites.get(1));
    }
}
```

输出为： Amazon

# ArrayList类

**计算大小**：如果要计算 ArrayList 中的元素数量可以使用 size() 方法

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites.size());
    }
}
```

输出为： 4

# ArrayList类

迭代数组列表：可以使用 for 来迭代数组列表中的元素

```
import java.util.ArrayList;
public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add(“Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
          for (int i = 0; i < sites.size(); i++) {
            System.out.println(sites.get(i));
          }
    }
}
```

输出为：

```
Google
Amazon
Taobao
Weibo
```

# ArrayList类

迭代数组列表：也可以使用 for-each 来迭代元素

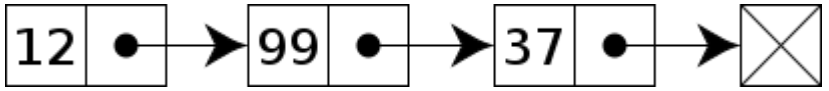
```
import java.util.ArrayList;
public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> sites = new ArrayList<String>();
        sites.add(“Google”);
        sites.add(“Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
         for (String i : sites) {
           System.out.println(i);
        }
    }
}
```

输出为：

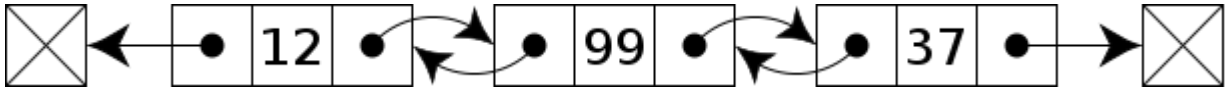
```
Google
Amazon
Taobao
Weibo
```

# LinkedList类

- LinkedList类的存储结构是双向链表，是常用的数据容器。它是一种线性表，但是**并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的地址**。链表可分为单向链表和双向链表。
- 一个单向链表包含两个值: 当前节点的值和一个指向**下一个**节点的链接。




- 一个双向链表包含三个值: 数值、**向后**的节点链接、**向前**的节点链接。




```
import java.util.LinkedList; // 引入 LinkedList 类
LinkedList<E> objectName = new LinkedList<E>(); // 普通创建方法
```

# ArrayList类 v.s. LinkedList类

- ArrayList
  - 必须开辟连续空间，查询**快**，增删**慢**。
  - 使用场景：用在频繁访问列表中的某一元素；只需要在列表末尾进行添加和删除操作。
- LinkedList
  - 无需开辟连续空间，查询慢，增删快。
  - 使用场景：需要频繁的在列表开头、中间或指定位置进行添加和删除元素操作。

# LinkedList类

实例：

```
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Taobao, Weibo]

# LinkedList类

在列表中间添加元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        // 使用 add (index, element) 在中间添加元素
        sites.add (1, "Wiki");
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Wiki, Taobao]

# LinkedList类

在列表开头添加元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        // 使用 addFirst() 在头部添加元素
        sites.addFirst("Wiki");
        System.out.println(sites);
    }
}
```

输出为： [Wiki,Google, Amazon, Taobao]

# LinkedList类

在列表结尾添加元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        // 使用 addLast() 在尾部添加元素
        sites.addLast("Wiki");
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Taobao,Wiki]

# LinkedList类

在列表开头移除元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
       sites.add("Weibo");
        // 使用 removeFirst() 移除头部元素
        sites.removeFirst();
        System.out.println(sites);
    }
}
```

输出为： [Amazon, Taobao,Weibo]

# LinkedList类

在列表结尾移除元素：

```
// 引入 LinkedList 类
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
       sites.add("Weibo");
        // 使用 removeLast() 移除尾部元素
        sites.removeLast();
        System.out.println(sites);
    }
}
```

输出为： [Google,Amazon, Taobao]

# LinkedList类

获取列表开头的元素：

*使用 getFirst() 获取头部元素*

获取列表结尾的元素：

*使用 getLast() 获取尾部元素*

# LinkedList类

**迭代元素**：我们可以使用 for 来迭代列表中的元素

```
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        for (int size = sites.size(), i = 0; i < size; i++) {
            System.out.println(sites.get(i));
        }
    }
}
```

输出为：

```
Google
Amazon
Taobao
Weibo
```

# LinkedList类

**迭代元素**：也可以使用 for-each 来迭代元素：

```
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> sites = new LinkedList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Weibo");
        for (String i : sites) {
            System.out.println(i);
        }
    }
}
```

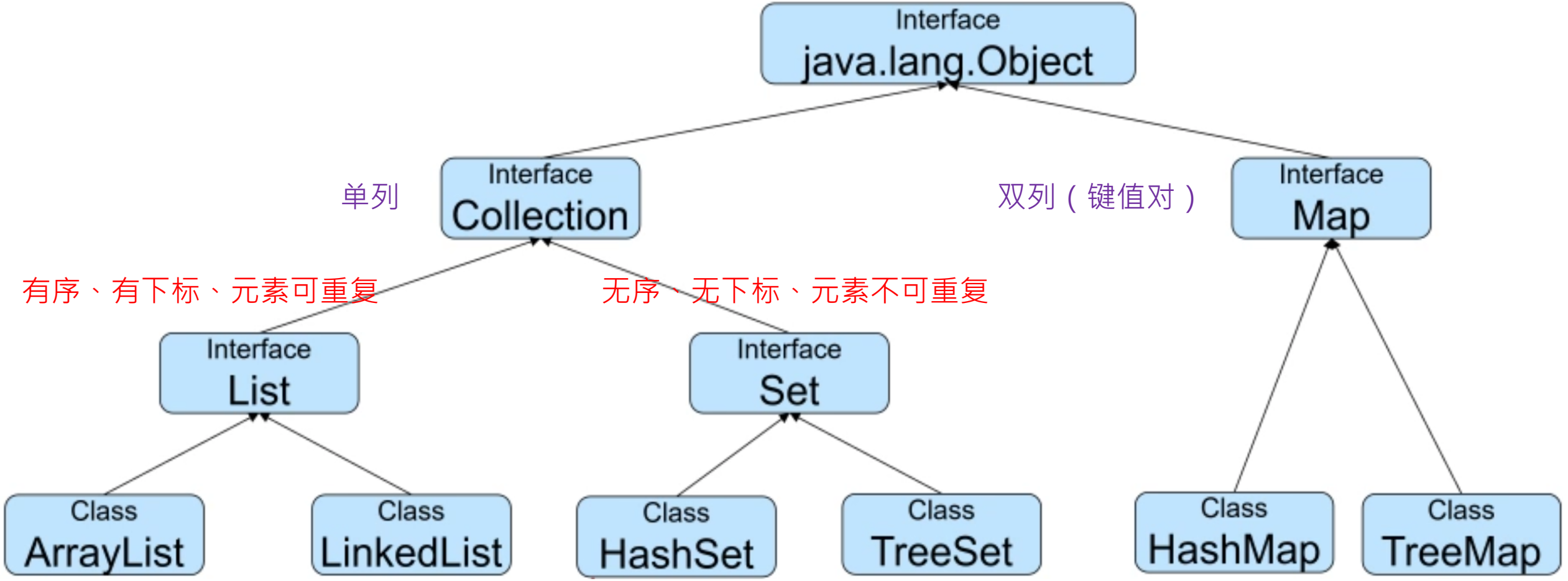
输出为：

```
Google
Amazon
Taobao
Weibo
```

# 互动小问题

在列表中间添加元素时，是否永远都是LinkedList比ArrayList快？

# ArrayList类 v.s. LinkedList类小总结




特点：底层数据结构是数组
查询快，增删慢
特有功能：无

特点：底层数据结构是链表
查询慢，增删快
特有功能：添加 **addFirst() addLast()**
删除 **removeFirst() removeLast()**
获取 **getFirst() getLast()**

# Set集合

- Set接口定义了一个无序的对象集合，特点：无序、无下标、元素不可以重复。

```
import java.util.HashSet; // 引入 HashSet 类
HashSet<E> sites = new HashSet<E>();
```

# 互动小问题

- 遍历HashSet集合时，能否使用for循环？

# HashSet类

**添加元素**：HashSet 类提供类很多有用的方法，添加元素可以使用 add() 方法:

```
// 引入 HashSet 类
import java.util.HashSet;
public class HashsetTest {
    public static void main(String[] args) {
    HashSet<String> sites = new HashSet<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        sites.add("Amazon");  // 重复的元素不会被添加
        System.out.println(sites);
    }
}
```

输出为： [Google, Amazon, Taobao, Zhihu]

# HashSet类

**判断元素是否存在**：我们可以使用 contains() 方法来判断元素是否存在于集合当中

```
// 引入 HashSet 类
import java.util.HashSet;
public class HashsetTest {
    public static void main(String[] args) {
    HashSet<String> sites = new HashSet<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        sites.add("Amazon");  // 重复的元素不会被添加
        System.out.println(sites.contains("Taobao"));
    }
}
```

输出为：　true

# HashSet类

**删除元素:**

我们可以使用 remove() 方法来删除集合中的元素:

```
sites.remove("Taobao"); // 删除元素，删除成功返回 true，否则为 false
```

删除集合中所有元素可以使用 clear 方法：

```
sites.clear();
```

# HashSet类

**计算大小:**

如果要计算 HashSet 中的元素数量可以使用 size() 方法：

```
sites.size()
```

# HashSet类

迭代 **HashSet**：可以使用 for-each 来迭代 HashSet 中的元素。

```
// 引入 HashSet 类
import java.util.HashSet;
public class HashsetTest {
    public static void main(String[] args) {
    HashSet<String> sites = new HashSet<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        sites.add("Amazon"); // 重复的元素不会被添加
        for (String i : sites) {
            System.out.println(i);
        }
    }
}
```

输出为：

```
Google
Amazon
Taobao
Zhihu
```

# Map集合

- Map接口存储的内容是键值对(key-value)映射，特点：无序，无下标，键不可重复，值可重复。
- key与value类型可以相同也可以不同。

```
import java.util.HashMap; // 引入 HashMap 类

HashMap<Integer, String> Sites = new HashMap<Integer, String>();
```

# HashMap

**添加元素：** HashMap 类提供了很多有用的方法，添加键值对(key-value)可以使用 put() 方法:

```
// 引入 HashMap 类
import java.util.HashMap;
public class HashMapTest {
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> Sites = new HashMap<Integer, String>();
        // 添加键值对
        Sites.put(1, "Google");
        Sites.put(2, “Amazon");
        Sites.put(3, "Taobao");
        Sites.put(4, "Zhihu");
        System.out.println(Sites);
    }
}
```

输出为： {1=Google, 2=Amazon, 3=Taobao, 4=Zhihu}

# HashMap

**删除元素:**

我们可以使用 remove(key) 方法来删除 key 对应的键值对(key-value):

```
Sites.remove(4);
```

删除集合中所有元素可以使用 clear 方法：

```
sites.clear();
```

# HashMap

**访问元素**：我们可以使用 get(key) 方法来获取 key 对应的 value:

```
// 引入 HashMap 类
import java.util.HashMap;
public class HashMapTest {
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> Sites = new HashMap<Integer, String>();
        // 添加键值对
        Sites.put(1, "Google");
        Sites.put(2, “Amazon");
        Sites.put(3, "Taobao");
        Sites.put(4, "Zhihu");
        System.out.println(Sites.get(3));
    }
}
```

输出为： Taobao

# HashMap

**计算大小:**

如果要计算 HashMap 中的元素数量可以使用 size() 方法：

```
sites.size()
```

# HashMap

迭代 **HashMap**：可以使用 for-each 来迭代 HashMap 中的元素。

```
// 引入 HashMap 类
import java.util.HashMap;
public class HashMapTest {
    public static void main(String[] args) {
        // 创建 HashMap 对象 Sites
        HashMap<Integer, String> Sites = new HashMap<Integer, String>();
        // 添加键值对
        Sites.put(1, "Google");
        Sites.put(2, “Amazon");
        Sites.put(3, "Taobao");
        Sites.put(4, "Zhihu");
```

# HashMap

**迭代 HashMap**：可以使用 for-each 来迭代 HashMap 中的元素。

```
        // 输出 key 和 value（如果你只想获取 key，可以使用 keySet() 方法）
        for (Integer i : Sites.keySet()) {
            System.out.println("key: " + i + " value: " + Sites.get(i));
        }
        // 输出 value 值（如果你只想获取 value，可以使用 values() 方法）
        for(String value: Sites.values()) {
          System.out.print(value + ", ");
        }
    }
}
```

输出为：

```
key: 1 value: Google
key: 2 value: Amazon
key: 3 value: Taobao
key: 4 value: Zhihu
Google, Amazon, Taobao, Zhihu,
```

# 策略模式

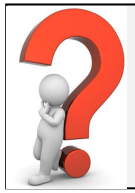
# 策略模式

- **策略模式（Strategy Pattern）**

在有多种算法相似的情况下，使用多个 if-else 语句会使代码变得复杂和难以维护，而策略模式允许策略随着对象改变而改变。

- 很多时候我们可以有多种策略来实现目的
  - 诸葛亮的锦囊妙计，每一个锦囊就是一个策略，可以视情况使用不同策略
  - 旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略
  - **...**

# 策略模式



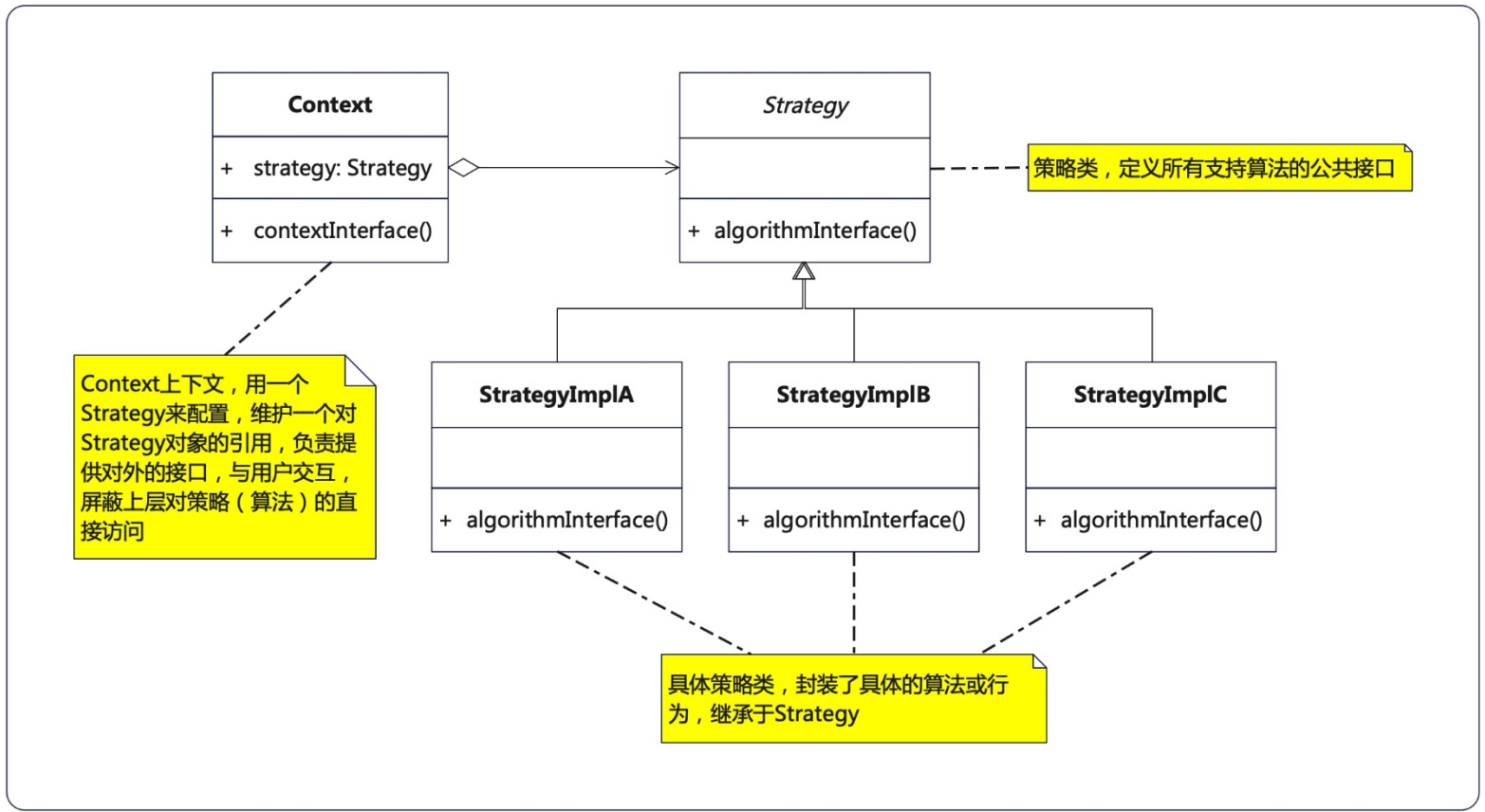
如何让算法和对象分开来，使得算法可以独立于使用它的客户而变化？

定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换，从而使得算法可独立于使用它的客户而变化。并将逻辑判断移到Client中去（即由客户端决定在什么情况下使用什么具体策略）。

# 策略模式（Strategy Pattern）

策略模式涉及到三个角色：

- 抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略（ConcreteStrategy）角色：包装了相关的算法或行为。
- 环境类/上下文类（Context）角色：持有一个Strategy的引用。



# 策略模式（Strategy Pattern）

例子：假设现在要设计一个卖书的电子商务网站，本网站可能对所有的高级会员提供每本20%的促销折扣；对中级会员提供每本10%的促销折扣；对初级会员没有折扣。

根据描述，折扣是根据以下的几个算法中的一个进行的：

- 算法一：对初级会员没有折扣。
- 算法二：对中级会员提供10%的促销折扣。
- 算法三：对高级会员提供20%的促销折扣。

# 策略模式（Strategy Pattern）

## 抽象折扣接口：

```
public interface MemberStrategy {
    /**
     * 计算图书的价格
     * @param booksPrice    图书的原价
     * @return    计算出打折后的价格
     */
    public double calcPrice(double booksPri
ce);
}
```

## 初级会员折扣类：

```
public class PrimaryMemberStrategy implements Member
Strategy {
    @Override
    public double calcPrice(double booksPrice) {
        System.out.println("对于初级会员的没有折扣");
        return booksPrice;
    }
}
```

# 策略模式（Strategy Pattern）

## 中级会员折扣类：

```
public class IntermediateMemberStrategy implements MemberStrategy {
    @Override
    public double calcPrice(double booksPrice) {
        System.out.println("对于中级会员的折扣为10%");
        return booksPrice * 0.9;
    }
}
```

## 高级会员折扣类：

```
public class AdvancedMemberStrategy implements MemberStrategy {
    @Override
    public double calcPrice(double booksPrice) {
        System.out.println("对于高级会员的折扣为20%");
        return booksPrice * 0.8;
    }
}
```

# 策略模式（Strategy Pattern）

网站类：

```
public class Network { //持有一个具体的策略对象
private MemberStrategy strategy;
/**
* 构造函数，传入一个具体的策略对象
* @param strategy 具体的策略对象
*/
    public Network(MemberStrategy strategy){
    this.strategy = strategy;
}
/**
  *计算图书的价格
* @param booksPrice 图书的原价
* @return 计算出打折后的价格
*/
  public double quote(double booksPrice){
    return this.strategy.calcPrice(booksPrice);
  }
}
```

# 策略模式（Strategy Pattern）

客户端类：

```
public class Client {
 public static void main(String[] args) {
    //选择并创建需要使用的策略对象
    MemberStrategy strategy = new AdvancedMemberStrategy();
    //创建环境
    Network network = new Network(strategy);
    //计算价格
    double quote = network.quote(300);
    System.out.println("图书的最终价格为：" + quote); }
}
```

为什么不直接写：double quote = strategy.calcPrice(300)?

为什么从环境类/上下文类里调用？

# 策略模式（Strategy Pattern）

**环境类/上下文类的作用：**

1. 需要明白行为（操作）的主体/**环境**是什么，要对现实世界做**正确**的抽象。

打折的主体/环境是Network，而不是Strategy

2.如果每个具体的策略都有一个需要执行的代码，可放于环境类，避免在每个策略里添加。

起到启下的作用

3.如有执行条件，也可放于环境类，保持客户端的简洁（只负责调用打折策略）。

起到承上的作用

**屏蔽高层模块对策略（算法）的直接访问，封装可能存在的变化**

# 策略模式（Strategy Pattern）

## 补充：

- 策略模式的**重心**：策略模式的重心不是如何实现算法，而是如何**组织、调用这些算法**，从而让程序结构更灵活，具有**更好的维护性和扩展性（满足开闭原则）**。

- 算法的**平等性**：策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是**相互独立的，相互之间是没有依赖的**。

- 运行时策略的**唯一性**：运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

# 策略模式

**策略模式总结：**

| 要素 | 描述 |
| --- | --- |
| **模式名<br>Pattern Name** | 策略模式（**Strategy Pattern**） |
| **目的<br>Intent** | 在有多种算法相似的情况下，避免多个 if-else 语句所带来的复杂和难以维护。 |
| **问题<br>Problem** | 如何让算法和对象分开来，使得算法可以独立于使用它的客户而变化？ |
| **解决方案<br>Solution** | 定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换，从而使得算法可独立于使用它的客户而变化。 |
| **效果<br>Consequence** | 优点：1、算法可以自由切换。 2、避免使用多重条件判断。 3、方便拓展和增加新的算法。<br>缺点: 1、如果备选的策略很多的话，那么对象的数目就会很可观。 2、客户端必须知道所有的策略类，并自行决定使用哪一个策略类。 |

# 互动小问题

## 工厂模式 vs. 策略模式

场景：飞机有很多种，有一个工厂专门负责生产各种需求的飞机。

**工厂模式的关注点：**
1）根据你给出的目的来生产不同用途的飞机，例如要运人，那么工厂生产客机，要运货就生产货机。
2）即根据你给出一些属性来生产不同行为的一类对象。

客户给它一个选择，它来帮客户创建一个对象。

**关注对象的创建：创建型模式**

**策略模式的关注点：**
1）用工厂生产的飞机来做对应的事情，例如用货机来运货，用客机来运人。
2）即根据你给出对应的对象来执行对应的方法。

客户给它一个对象，它来帮客户做对应的选择。

**关注行为的选择：行为型模式**

# 互动小问题

## 重看面向对象的思想

- × 可能带来额外的时间和空间开销。
- ✓ 面向更高的逻辑抽象层，可减少耦合，可封装变化，尽量避免错误蔓延。

# 迭代器模式

# 互动小问题

如果要访问聚合对象中的各个元素，比如ArrayList, 可以用for来遍历，可以用for each 遍历。

1. 这种方式是否满足开闭原则？
2. 这种方式是否满足单一职责原则？

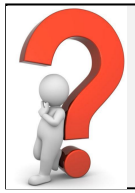
# 迭代器模式

- **迭代器模式（Iterator Pattern）**

在客户访问类与聚合类之间插入一个迭代器，这**分离了聚合对象与其遍历行为，对客户也隐藏了其内部细节**，且满足**“单一职责原则”**和**“开闭原则”**。

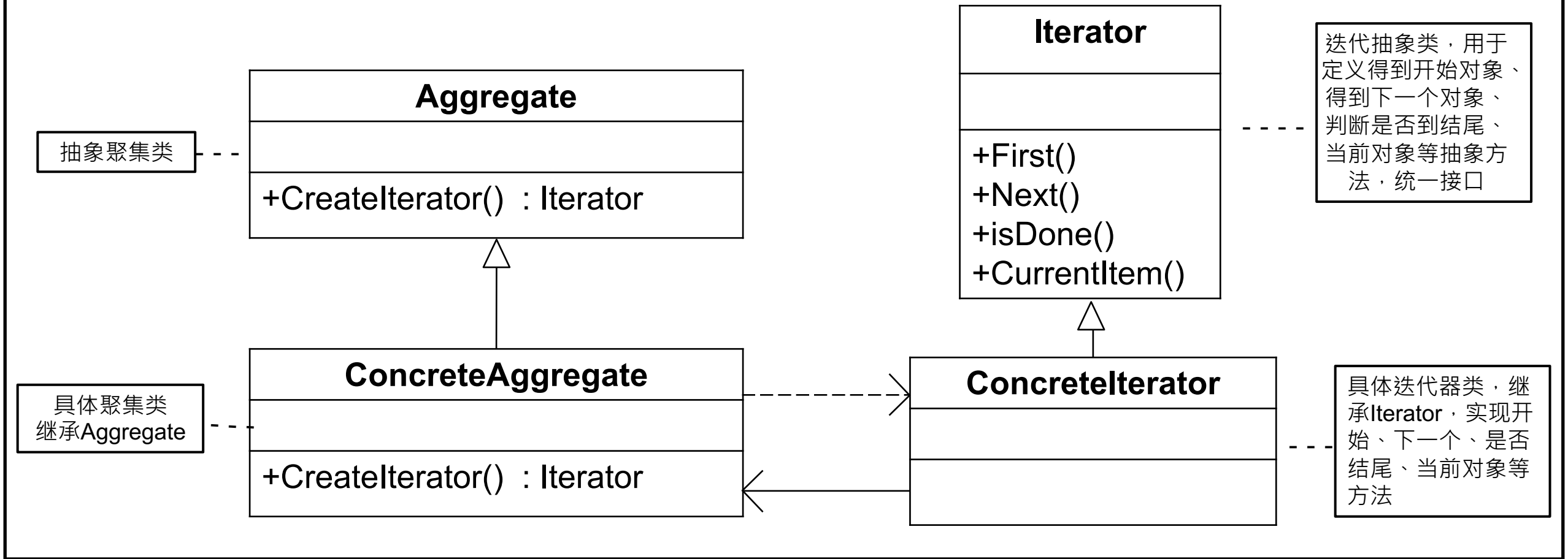
- 迭代器模式在生活中应用的比较广泛
  - 物流系统中的传送带，不管传送什么物品，都打包成一个个箱子，并且有一个统一的二维码，在分发时只需要一个个检查发送的目的地即可。
  - 乘坐交通工具，都是统一刷卡或者刷脸进站，而不需要关心是男性还是女性、是残疾人还是正常人等信息。
  - **...**

# 迭代器模式

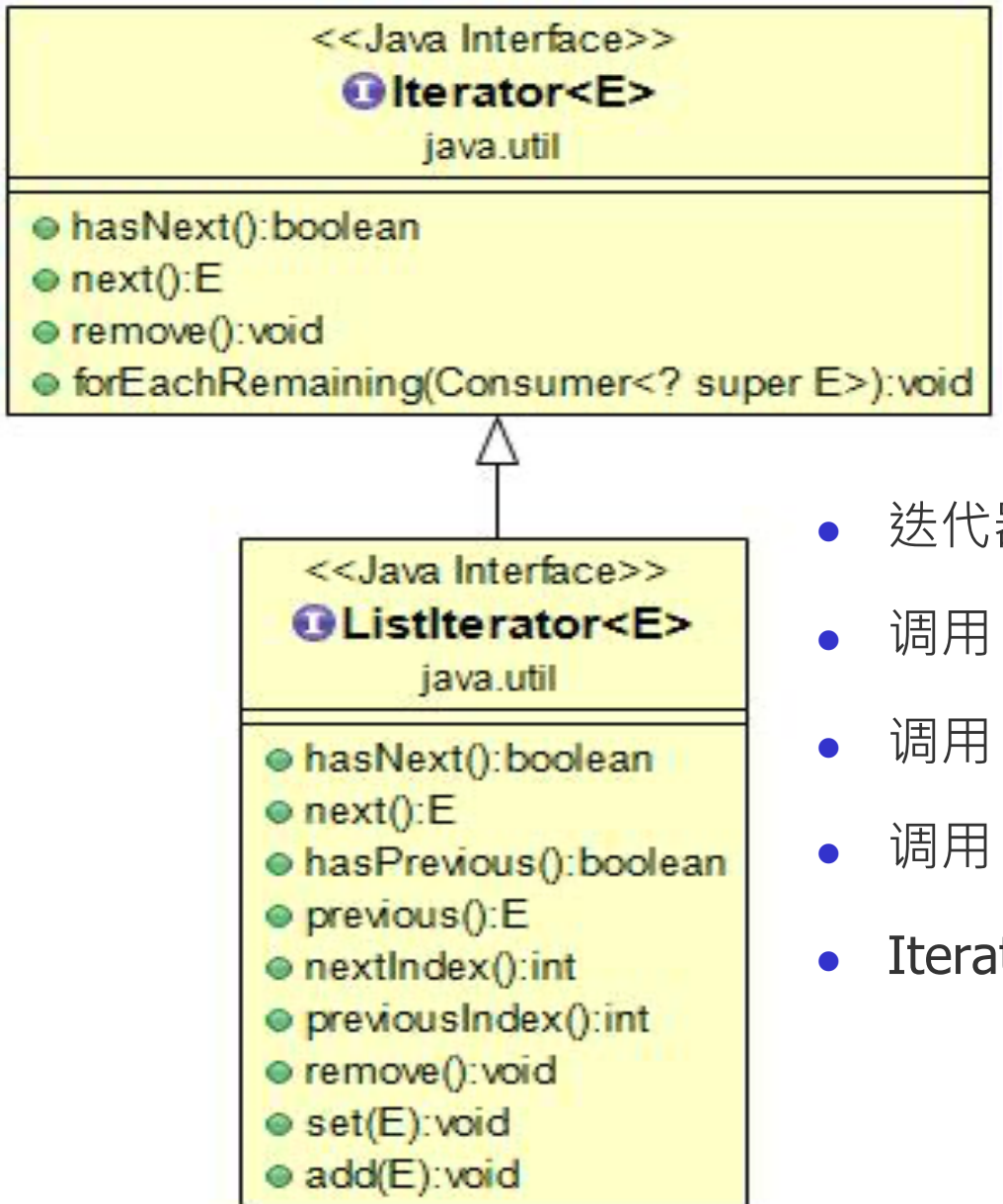


如何将聚合对象与其遍历行为分离开？

迭代器模式（Iterator Pattern）结构图



# 迭代器模式




- 迭代器 it 的三个基本操作是 next 、hasNext 和 remove。
- 调用 it.next() 会返回迭代器的下一个元素，并且更新迭代器的状态。
- 调用 it.hasNext() 用于检测集合中是否还有元素。
- 调用 it.remove() 将迭代器返回的元素删除。
- Iterator 类位于 java.util 包中，使用前需要引入它，语法格式如下：

```
import java.util.Iterator; // 引入 Iterator 类
```

# 迭代器模式

实例：

```
// 引入 ArrayList 和 Iterator 类
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorTest {
    public static void main(String[] args) {
        // 创建集合
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Amazon");
        sites.add("Taobao");
        sites.add("Zhihu");
        // 获取迭代器
        Iterator<String> it = sites.iterator();
        // 输出集合中的第一个元素
        System.out.println(it.next());
    }
}
```

输出为：

```
Google
```

# 迭代器模式

**循环集合元素**:让迭代器 it 逐个返回集合中所有元素最简单的方法是使用 while 循环：

```
while(it.hasNext()) {
        System.out.println(it.next());
}
```

输出为：

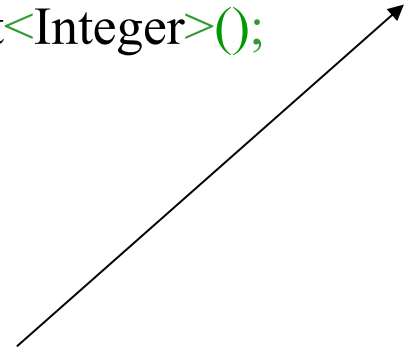
```
Google
Amazon
Taobao
Weibo
```

# 迭代器模式

删除元素:要删除集合中的元素可以使用 remove() 方法。

以下实例我们删除集合中小于 10 的元素 ：

```
// 引入 ArrayList 和 Iterator 类
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorTest {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(12);
        numbers.add(8);
        numbers.add(2);
        numbers.add(23);
        Iterator<Integer> it = numbers.iterator();
```



```
while(it.hasNext()) {
        Integer i = it.next();
        if(i < 10) {
          // 删除小于 10 的元素
            it.remove();
        }
    }
    System.out.println(numbers);
}}
```

输出为： [12, 23]

# 互动小问题

迭代器模式是什么型模式？为什么？

# 迭代器模式

迭代器模式总结：

| 要素 | 描述 |
| --- | --- |
| **模式名<br>Pattern Name** | 迭代器模式（**Iterator Pattern**） |
| **目的<br>Intent** | 提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。 |
| **问题<br>Problem** | 1、访问一个聚合对象的内容而无须暴露它的内部表示。 2、需要为聚合对象提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。 |
| **解决方案<br>Solution** | 把在元素之间游走的责任交给迭代器，而不是聚合对象。 |
| **效果<br>Consequence** | 优点：1）它支持以不同的方式遍历一个聚合对象。 2）迭代器简化了聚合类。 3）在同一个聚合对象上可以有多个遍历。 4）在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。 |