

OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
 - Project 6: 6月12日晚9点 -> 6月5日晚9点
 - Project 7: 6月12日晚9点
 - 本周/下周上机课上课
 - 考试时间（暂定）：6月20日，随堂时间

- 复习
 - 继承
 - 代码复用
 - 向上转换, 多态(父类与子类的类型转换)
 - 接口
 - ~~代码复用~~
 - 向上转换, 多态(多个父接口)
 - 内部类
 - 代码复用
 - 向上转换, 多态(灵活实现多继承)

- 复习

- 内部类

- 定义在一个类的内部

```
class Outer{  
    ...  
    class Inner{  
        ...  
    }  
    ...  
}
```

```
public class Parcel{  
    class Contents{  
        private int i = 11;  
        public int value() {return i;}  
    }  
    class Destination{  
        private String label;  
        Destination(String r) {label = r;}  
        String readLabel() { return label;}  
    }  
    public Destination to(String s){  
        return new Destination(s);  
    }  
    public Contents contents(){  
        return new Contents();  
    }  
    public void ship(String dest){  
        Contents c = new Contents();  
        Destination d = new Destination(dest);  
        System.out.println(d.readLabel());  
    }  
  
    public static void main(String []args){  
        Parcel p = new Parcel();  
        Parcel.Destination d = p.to("Tasmania");  
        Parcel.Contents c = p.contents();  
    }  
}
```

- 复习

- 内部类

- 每个内部类对象包含的有一个外部类对象的引用
 - OuterClassName.this
 - 创建内部类
 - OuterClassObject.new
 - 在外部类的非静态方法中可以直接创建（默认外部类对象是this)

```
public class Parcel{
    class Contents{
        private int i = 11;
        public int value() {return i;}
    }
    class Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }

    public Destination to(String s){
        return new Destination(s);
    }
    public Contents contents(){
        return new Contents();
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Parcel.Destination d = p.new Destination("T");
        Parcel.Contents c = p.new Contents();
    }
}
```

- 复习

- 匿名内部类

- 没有名字的内部类
 - 必须继承某个类, 或实现某个接口

```
public class Parcel{

    public Contents contents(){
        return new Contents() {
            // anonymous inner class definition
            private int i = 11;
            public int value() {return i;}
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```



```
public class Parcel{
    class PContents implements Contents{
        private int i = 11;
        public int value() {return i;}
    }

    public Contents contents(){
        return new PContents() ;
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

• 复习

– 通过内部类灵活实现多继承

```
interface A { }
interface B { }
class X implements A, B { }

class Y implements A{
    B makeB() {
        return new B();
    }
}

public class Test{
    static void takeA(A a) {}
    static void takeB(B b) {}
    public static void main(String []args){
        X x = new X();
        Y y = new Y();
        takeA(x); takeB(x);
        takeA(y); takeB(b.makeB());
    }
}
```

```
class A { }
abstract class B { }
// class X implements A, B { }
// won't compile

class Y extends A{
    B makeB() {
        return new B();
    }
}

public class Test{
    static void takeA(A a) {}
    static void takeB(B b) {}
    public static void main(String []args){
        Y y = new Y();
        takeA(y); takeB(b.makeB());
    }
}
```


OOP with Java

- 容器简介
- Collection
 - List, Set, Queue
- Map
- Collection and Iterator

容器简介

- 如何将对象组织起来？

```
int a = 0;
```

```
int b = 0;
```

```
...
```

```
int z = 0;
```

```
MyType m_a = new MyType();
```

```
MyType m_b = new MyType();
```

```
...
```

```
MyType m_c = new MyType();
```

容器简介

- 数组

```
int [ ] a = new int[]{1,2,3};
```

```
MyType [ ] b = new MyType[3];
```

```
MyType [ ] c = new MyType[3] {  
    new MyType(),  
    new MyType(),  
    new MyType() };
```

长度不可变

1. 无法添加和删除数组元素
2. 数组元素之间的关系? (Set)

容器简介

- 容器
 - 提供更灵活的组织对象的方式
 - 动态添加, 删除
 - 例如
 - List, Set, Queue
 - Map
 - 位于包java.util中



Item1

Item2

ItemN

List: 一系列有序的对象
(数组,链表)

```
import java.util.*;
```

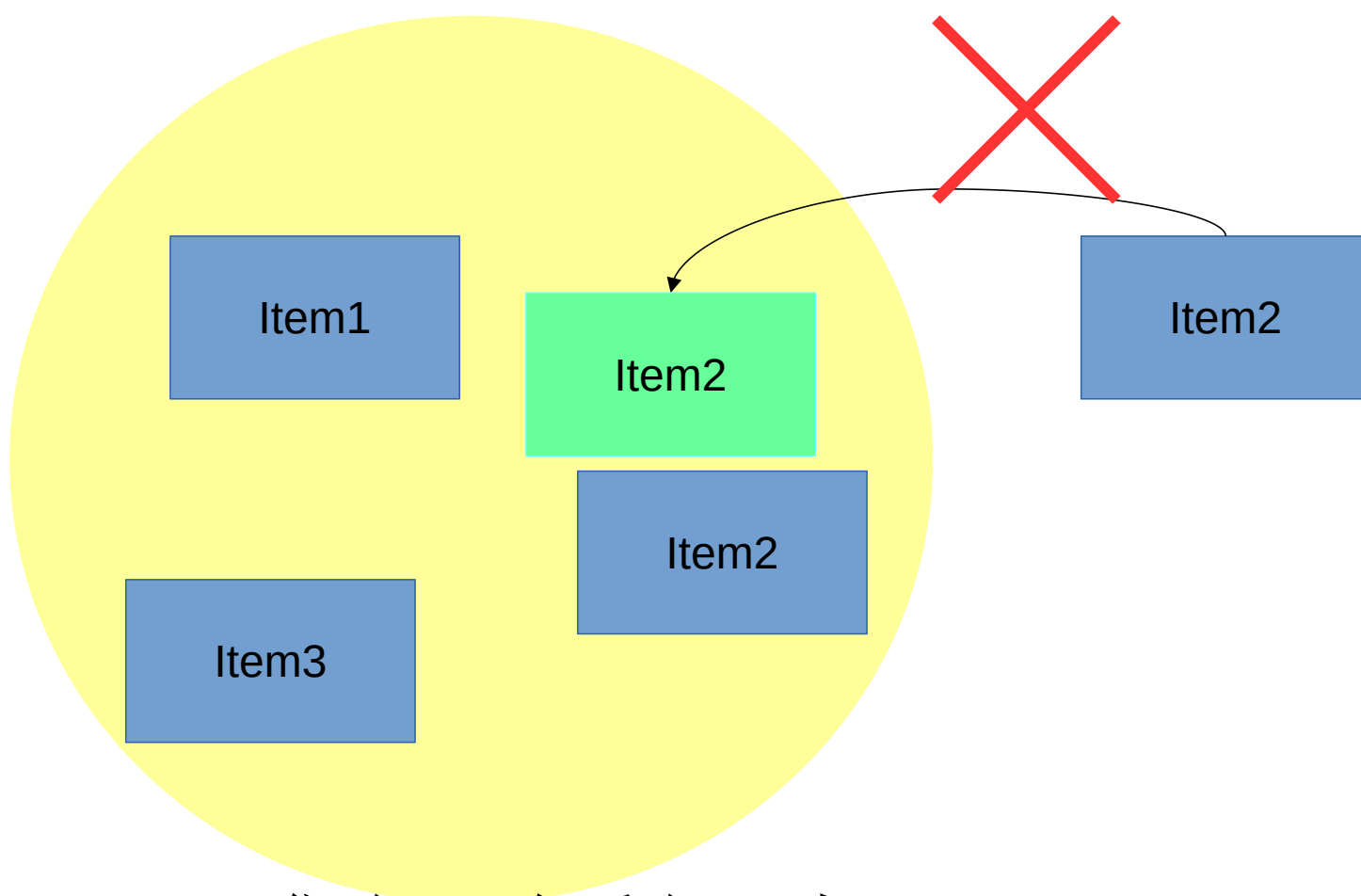
```
ArrayList a = new ArrayList();  
LinkedList b = new LinkedList();
```

```
//Implement List interface
```

```
List c = a;
```

```
List d = b;
```

List接口



Set: 集合(没有重复元素)

```
import java.util.*;
```

```
HashSet a = new HashSet();
```

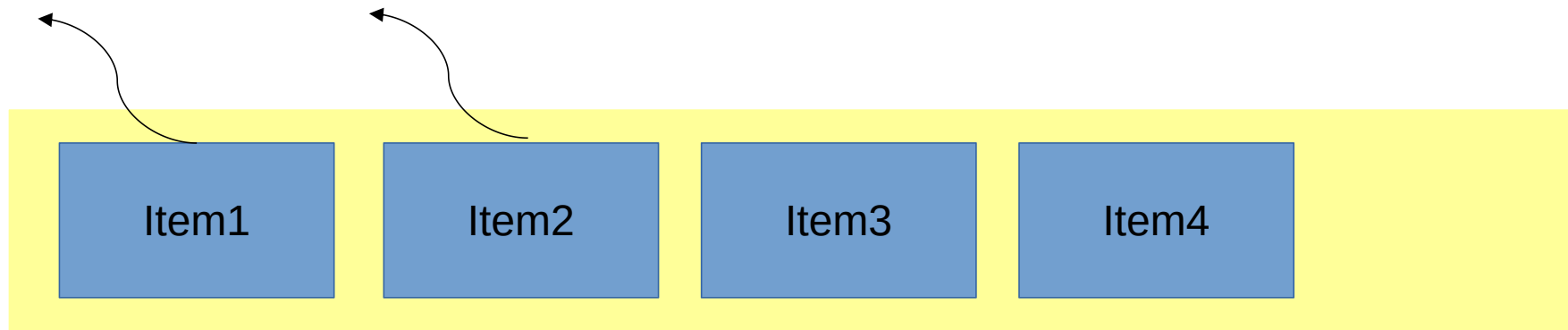
```
TreeSet b = new TreeSet();
```

```
//implement Set interface
```

```
Set c = a;
```

```
Set d = b;
```

Set接口



Queue: 队列

- enqueue (进队)
- dequeue (出队)
- 先进先出(First in, first out)
- 应用: 任务调度

```
import java.util.*;
```

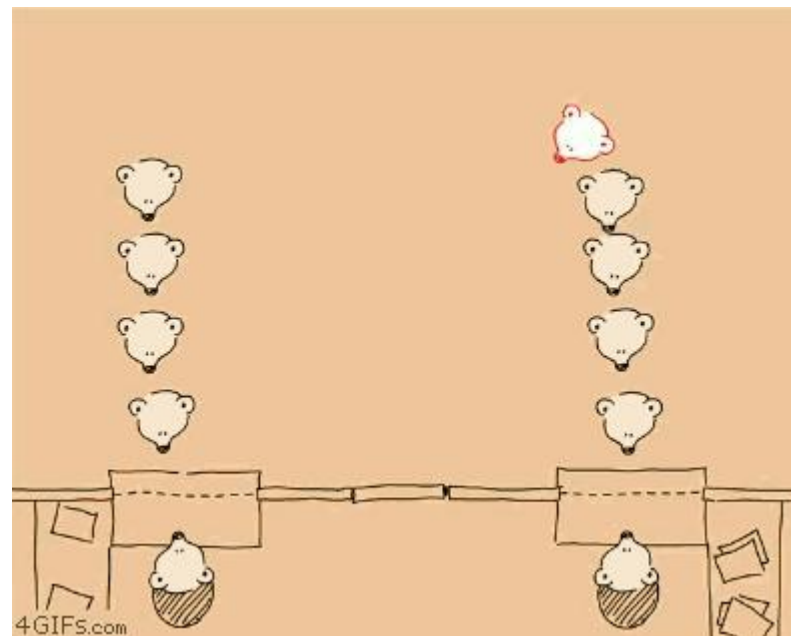
```
LinkedList a = new LinkedList();  
PriorityQueue b = new PriorityQueue();
```

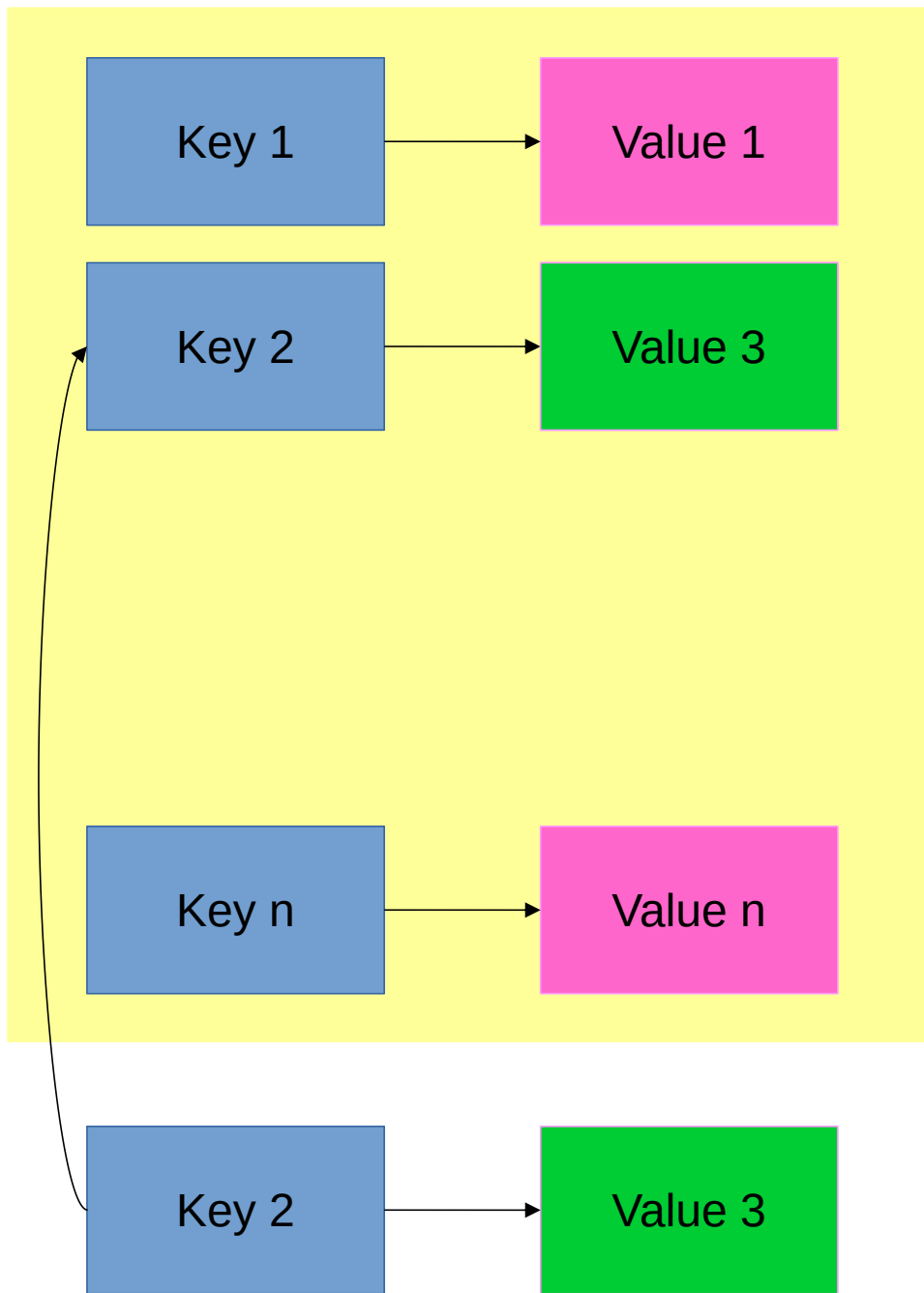
```
//Implement Queue interface Queue接口
```

```
Queue c = a;
```

```
Queue d = b;
```

- 队列
 - 先进先出(First in, first out)
 - 先来先服务 (First come, first serve)





Map:

- Key-value 对
- Key不重复
- value 可以重复
- 应用: 单词出现次数

```
import java.util.*;
```

```
//Implement Map interface
```

```
HashMap a = new HashMap();
```

Map接口

容器简介

- 泛型(generic) 与类型安全的容器
 - 容器可以存放的类型为Object
 - 任何类型的对象都能放入容器
 - 容器的类型只能在运行时确定

```
class Apple {  
    private static long counter;  
    private final long id = counter++;  
    public long id() { return id; }  
}  
class Orange { }
```

```
public class ApplesAndOrangesWithoutGenerics {  
    public static void main(String[] args) {  
        ArrayList apples = new ArrayList();  
        for(int i = 0; i < 3; i++)  
            apples.add(new Apple());  
  
        // Not prevented from adding an Orange to apples:  
        apples.add(new Orange());  
        for(int i = 0; i < apples.size(); i++)  
            ((Apple)apples.get(i)).id();  
        // Orange is detected only at run time  
    }  
}
```

容器简介

- 类型安全的容器
 - 定义容器为只能存放某种类型的对象
 - 编译时确定类型
- 泛型编程(generic)

```
class Apple {  
    private static long counter;  
    private final long id = counter++;  
    public long id() { return id; }  
}  
class Orange { }
```

```
public class ApplesAndOrangesWithGenerics {  
    public static void main(String[] args) {  
        ArrayList<Apple> apples = new ArrayList<Apple>();  
        for(int i = 0; i < 3; i++)  
            apples.add(new Apple());  
  
        // Compile error!  
        // apples.add(new Orange());  
        for(int i = 0; i < apples.size(); i++)  
            apples.get(i).id();  
        for(Apple c: apples)  
            System.out.println(c.id());  
    }  
}
```

容器简介

- 类型安全的容器
 - Upcasting 适用

```
class GrannySmith extends Apple {}  
class Gala extends Apple {}  
class Fuji extends Apple {}  
class Braeburn extends Apple {}
```

```
public class GenericsAndUpcasting {  
    public static void main(String[] args) {  
        ArrayList<Apple> apples = new ArrayList<Apple>();  
  
        apples.add(new GrannySmith());  
        apples.add(new Gala());  
        apples.add(new Fuji());  
        apples.add(new Braeburn());  
        for(Apple c : apples)  
            System.out.println(c);  
    }  
}
```

容器简介

- 类型安全的容器
 - 不能指定基本类型
 - 使用基本类型的wrapper
 - Autoboxing and unboxing

```
import java.util.*;  
  
// compile error  
// ArrayList<int> a = new ArrayList<int>();  
ArrayList<Integer> a = new ArrayList<Integer>();  
For (int i = 0; i < 10; ++i)  
    a.add(i); //autoboxing
```


容器简介

- 容器接口
 - Collection接口: 用于存放一组对象
 - List接口: 对象按照插入顺序排列容器中的对象
 - Set接口: 容器中不能有重复的对象
 - Queue接口: 按"队列"规则插入/删除对象
 - Map接口
 - 用于存放一组 “键-值对” (key-value pair)
 - key的类型, value的类型
 - 按照key查找对应的value
 - 也称为dictionary, associative array

容器简介

```
import java.util.*;  
  
//List is an interface  
List<Apple> a = new ArrayList<Apple>();  
List<Apple> b = new LinkedList<Apple>();  
  
//Collection is an interface  
Collection<Apple> c = new ArrayList<Apple>();
```

容器简介

- 容器重写了toString()方法, 可以帮助可视化容器的内容

```
import java.util.*;  
  
ArrayList<String> a = new ArrayList<String>();  
a.add("rat");  
a.add("cat");  
a.add("dog");  
a.add("dog");  
System.out.println(a);
```

```

public class PrintContainers{
    static Collection fill(Collection<String> c){
        c.add("rat");
        c.add("cat");
        c.add("dog");
        c.add("dog");
    }
    static Map fill(Map<String, String> m){
        m.put("rat", "Fuzzy");
        m.put("cat", "Rags");
        m.put("dog", "Bosco");
        m.put("dog", "Spot");
    }
    public static void main(String [] args){
        System.out.println(fill(new ArrayList<String>()));
        System.out.println(fill(new LinkedList<String>()));

        System.out.println(fill(new HashSet<String>()));
        System.out.println(fill(new TreeSet<String>()));
        System.out.println(fill(new LinkedHashSet<String>()));

        System.out.println(fill(new HashMap<String, String>()));
        System.out.println(fill(new TreeMap<String, String>()));
        System.out.println(fill(new LinkedHashMap<String, String>()));
    }
}

```

```

[rat, cat, dog, dog] //ArrayList
[rat, cat, dog, dog] //LinkedList
[cat, dog, rat] // HashSet
[cat, dog, rat] // TreeSet
[rat, cat, dog] // LinkedHashSet
{cat=Fuzzy, dog=Spot, rat=Fuzzy} //HashMap
{cat=Fuzzy, dog=Spot, rat=Fuzzy} //TreeMap
{rat=Fuzzy, cat=Fuzzy, dog=Spot}
//LinkedHashMap

```

List:

- ArrayList 实现为数组
- LinkedList 实现为链表

Set/Map

- Hash: 实现为hash表, 查询较快
- Tree: 实现为查询树, 按顺序排列
- LinkedHashMap: 按照插入顺序排列

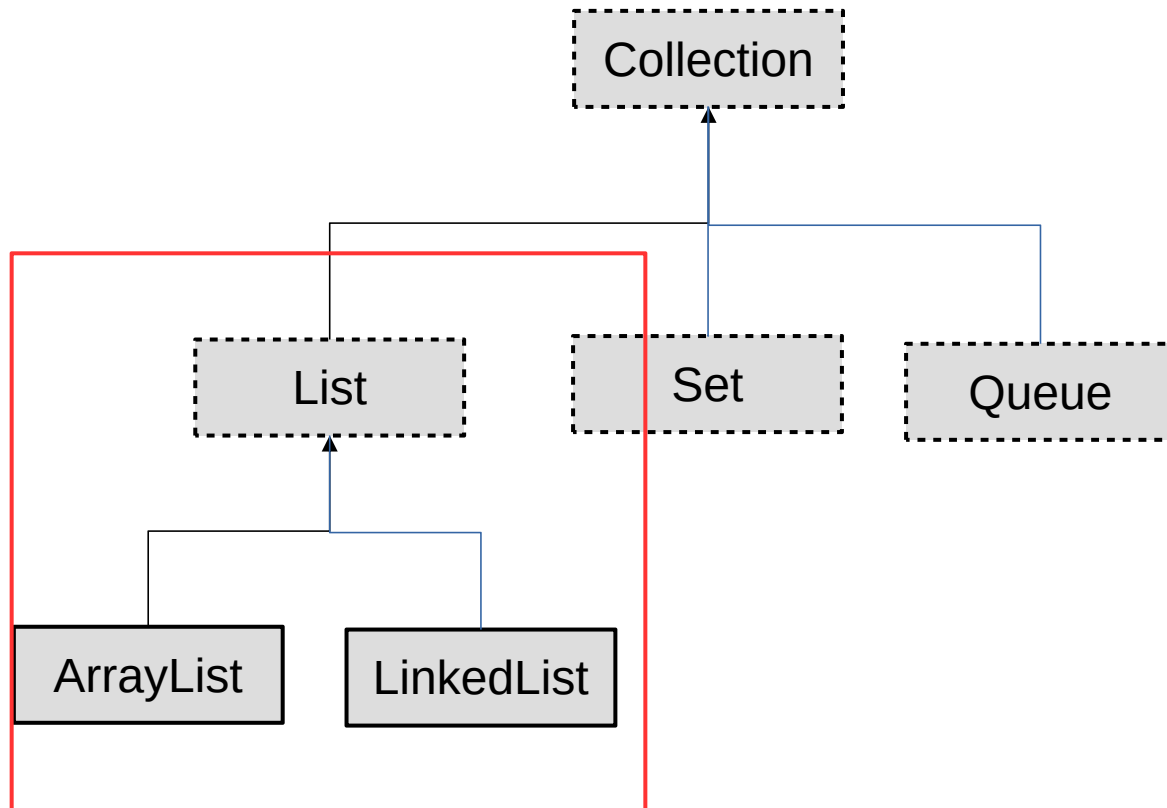
容器简介

- 总结
 - 容器类型
 - Collection: List, Set, Queue
 - Map
 - 类型安全的容器
 - `ArrayList<T> a = new ArrayList<T>();`
 - Upcasting and Autoboxing

List

- 两种类型的List
 - ArrayList
 - “可扩展数组”
 - 适用于随机访问, 插入删除较慢
 - LinkedList
 - 双向链表
 - 适用于顺序访问, 插入删除较快
 - 实现了Queue接口

List



List

- List 接口
 - add(): 添加元素
 - remove(): 删除元素
 - get(): 返回第i个位置的元素
 - size(): 返回元素数量
 - ...

List

- 构造函数

- ArrayList

```
ArrayList<E>();  
ArrayList<E>(int initialCapacity);  
ArrayList<E>(Collection<E> c);
```

- LinkedList

```
LinkedList<E>();  
LinkedList<E>(Collection<E> c);
```

```
import java.util.*;

ArrayList<String> a = new ArrayList<String>();
// 插入add(Object o)
a.add("rat"); a.add("cat"); a.add("dog"); a.add("dog");

// 查询contains (Object o)
System.out.println(a.contains("cat"));

// 删除remove(Object o) (若不在List中, 返回false, 否则返回true)
a.remove("dog"); a.remove("dag");

// 访问第i个元素: get(int)
a.get(0);

// 对象的数量: size()
a.size();

// 序号 indexOf
a.indexOf("cat");
```

```
import java.util.*;
```

```
ArrayList<String> a = new ArrayList<String>();
```

```
// 插入add(Object o)
```

```
a.add("rat"); a.add("cat"); a.add("dog"); a.add("dog");
```

```
// 子表 subList(int fromIndex, int toIndex)
```

```
List<String> sub = a.subList(2, 3);
```

```
// 是否为空 isEmpty()
```

```
System.out.println(a.isEmpty());
```

```
// 返回迭代器 iterator()
```

```
Iterator it = a.iterator();
```

```
// 返回List迭代器 listIterator()
```

```
ListIterator lit = a.listIterator();
```

```
// 转为数组
```

```
Object [] aarray = a.toArray();
```

```
....
```

List

- Iterator (迭代器)

- 通常需要访问/遍历Collection中的元素

```
ArrayList<String> a = new ArrayList<String>();
```

```
.....
```

```
a.get(i);
```

```
LinkedList<String> b = new LinkedList<String>();
```

```
.....
```

```
b.get(i);
```

```
Static void visit(List<String> ls) {
```

```
    ls.get(i);
```

```
    ....
```

```
}
```

- 缺点
 - 依赖于Collection接口
 - 其他没有实现Collection接口的类无法使用函数visit
- 解决方法iterator, 包含方法
 - next()
 - hasNext()
 - remove()
- Collection接口
 - iterator()
 - 返回该List的迭代器

List

- iterator()

```
ArrayList<String> a = new ArrayList<String>();  
a.add("rat");a.add("cat");a.add("dog");a.add("dog");  
Iterator<String> it = a.iterator();  
while(it.hasNext()){  
    String s = it.next();  
    System.out.println(s);  
}  
  
// identical to  
for(String i: a)  
    System.out.println(i);
```

- iterator

```
Import java.util.*
public class Iteration {
    public static void display(Iterator<String> it){
        while(it.hasNext()){
            String s = it.next();
            System.out.println(s);
        }
    }
    public static void main(String[]args){
        ArrayList<String> a = new ArrayList<String>();
        a.add("rat");a.add("cat");a.add("dog");a.add("dog");

        LinkedList<String> b = new LinkedList<String>(a);
        HashSet<String> c = new HashSet<String>(a);
        TreeSet<String> d = new TreeSet<String>(a);

        display(a.iterator());
        display(b.iterator());
        display(c.iterator());
        display(d.iterator());
    }
}
```

List

- ListIterator
 - List接口提供
 - 扩展了Iterator
 - 双向遍历
 - hasNext(), hasPrevious()
 - next(), previous()

List

- LinkedList
 - 实现List接口
 - 实现Queue接口
 - add(), remove(), element()
 - offer(), poll(), peek()
 - 提供更多的方法

List

- LinkedList方法

```
import java.util.*;
```

```
LinkedList<String> a = new LinkedList<String>();  
a.add("rat"); a.add("cat"); a.add("dog"); a.add("dog");
```

```
// 返回链表首元素 getFirst(), element(), 若链表为空则抛出异常  
a.getFirst(); a.element();
```

```
// 返回链表首元素 peek(), 若链表为空则返回null  
a.peek();
```

```
// 删除并返回链表首元素 removeFirst(), remove(), 若链表为空则抛出异常  
String s = a.remove();
```

```
// 删除并返回链表首元素 poll(), 若链表为空则返回null  
String s = a.poll();
```

List

- LinkedList方法

```
import java.util.*;
```

```
LinkedList<String> a = new LinkedList<String>();  
a.add("rat"); a.add("cat"); a.add("dog"); a.add("dog");
```

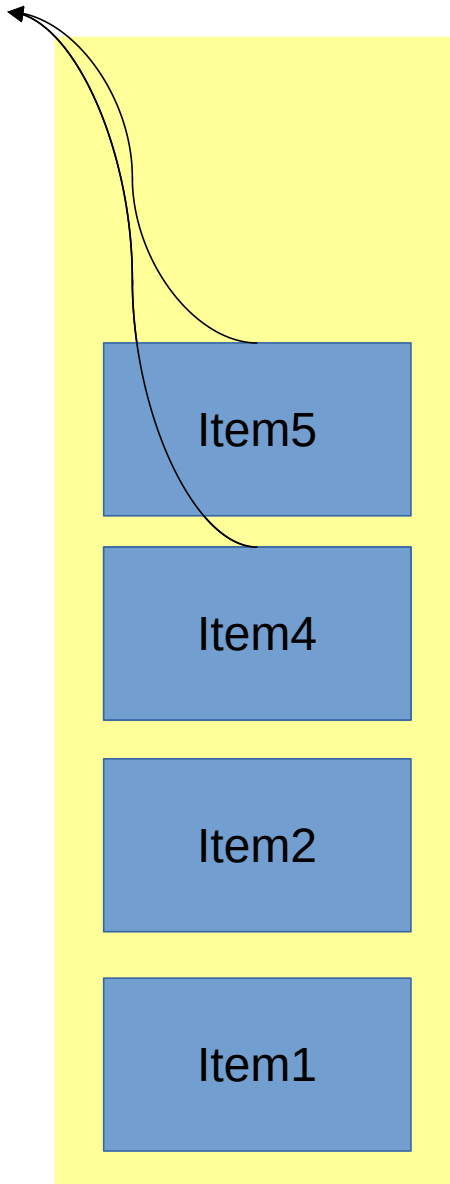
```
// 在链表头添加对象 addFirst()  
a.addFirst("tiger");
```

```
// 在链表尾添加对象 add(), addLast(), offer()  
a.add("cow"); a.addLast("cow");
```

List

- LinkedList应用: Stack
 - 后进先出(Last In First Out, LIFO)
 - push: 将一个对象入栈
 - pop: 从栈中取出一个元素: 按照LIFO原则

List



```
LinkedList<Item> s = new LinkedList<Item>();
```

```
s.push(item1);
```

```
s.push(item2);
```

```
s.push(item3);
```

```
s.pop();
```

```
s.push(item4);
```

```
s.push(item5);
```

```
s.pop();
```

```
s.pop();
```

List

```
import java.util.LinkedList;
public class Stack<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void push(T v) { storage.addFirst(v); }
    public T peek() { return storage.getFirst(); }
    public T pop() { return storage.removeFirst(); }
    public boolean empty() { return storage.isEmpty(); }
    public String toString() { return storage.toString(); }
}
```

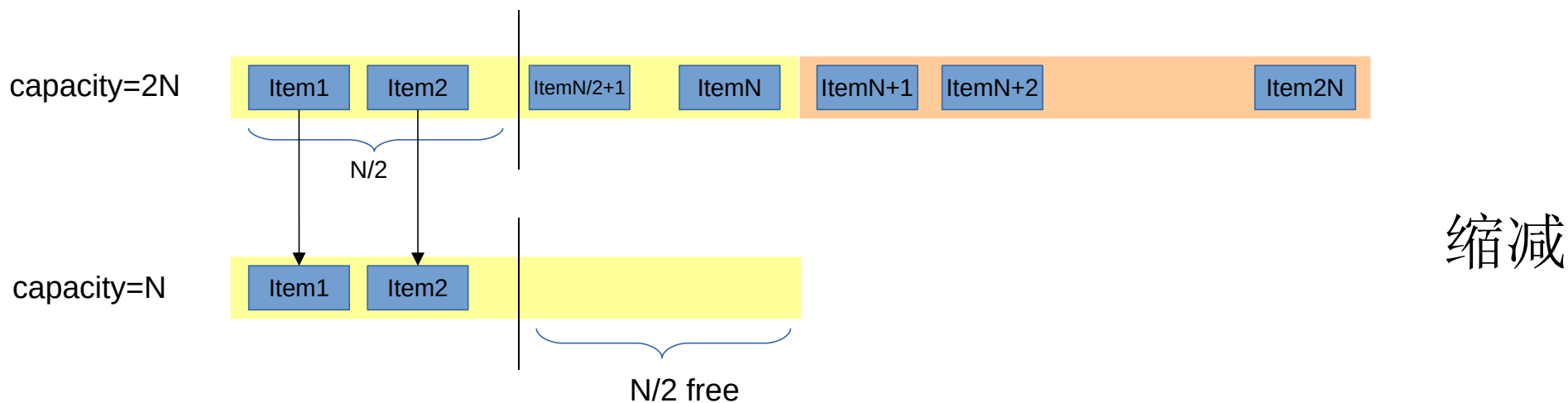
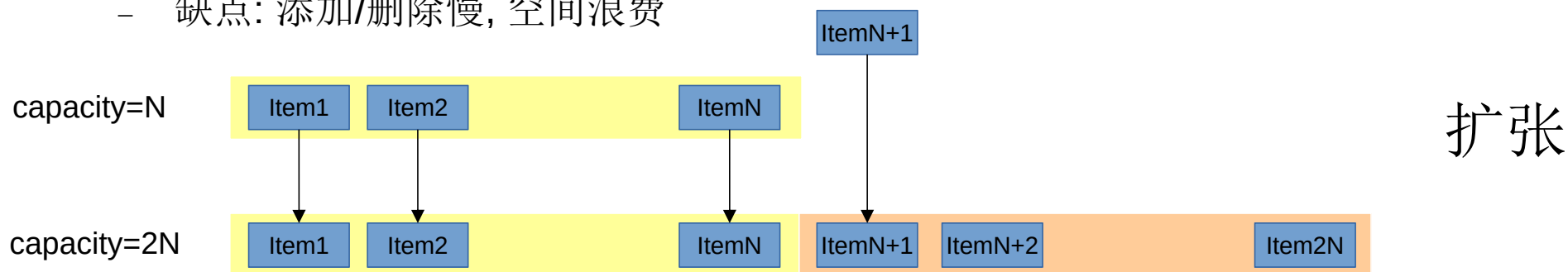
```
public class StackTest {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
    }
}
```

List

- Stack的应用
 - 上下文无关文法

- ArrayList

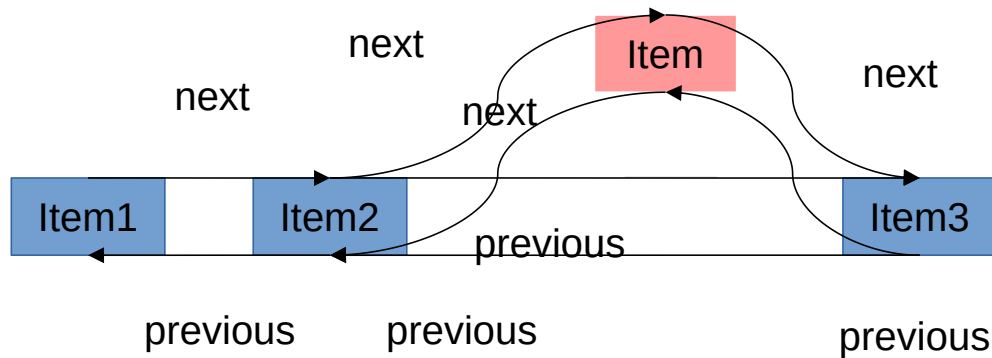
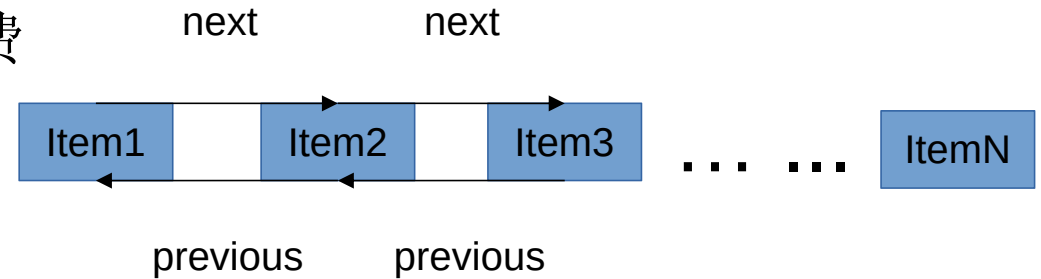
- 对象存储在数组中(可变长数组)
- 优点: 随机访问快
- 缺点: 添加/删除慢, 空间浪费



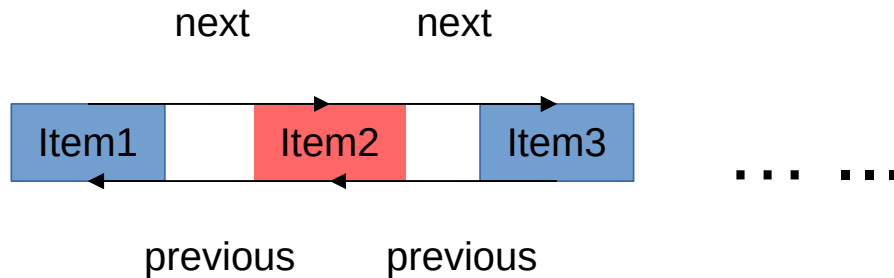
每次扩张或缩减数组长度时, 保证新的数组有一半的可用空间

• LinkedList

- 双向链表
- 优点: 添加/删除较快, 无空间浪费
- 缺点: 随机访问慢



插入

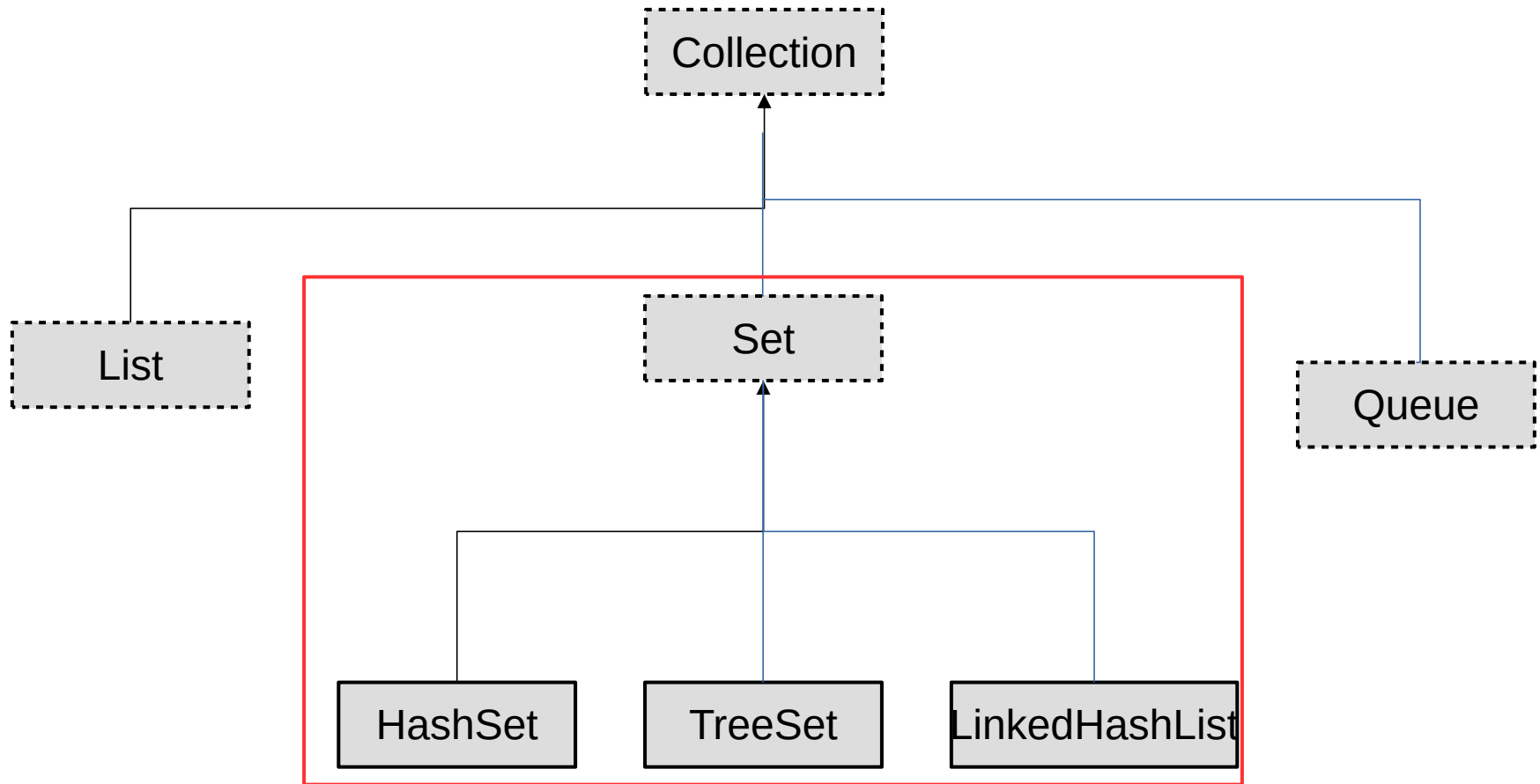


删除

List

- 总结
 - List接口
 - add(), remove(), get(), size(), indexOf()
 - ArrayList
 - 可变长度数组
 - LinkedList
 - 链表
 - 实现Queue接口
 - 迭代器 Iterator

Set



Set

- Set接口
 - 容器中不能出现重复的元素
 - 没有对 **Collection** 接口扩展
 - 三种主要实现
 - HashSet
 - TreeSet
 - LinkedHashSet

Set

- Set 接口
 - add(Object o), addAll(Collection<E> c)
 - remove(Object o), removeAll(Collection<E> c)
 - contains(Object o)
 - iterator()
 - size()
 - toArray()
 - ...

Set

- HashSet

- 特点: 快速 (插入, 删除, 查找), 无序

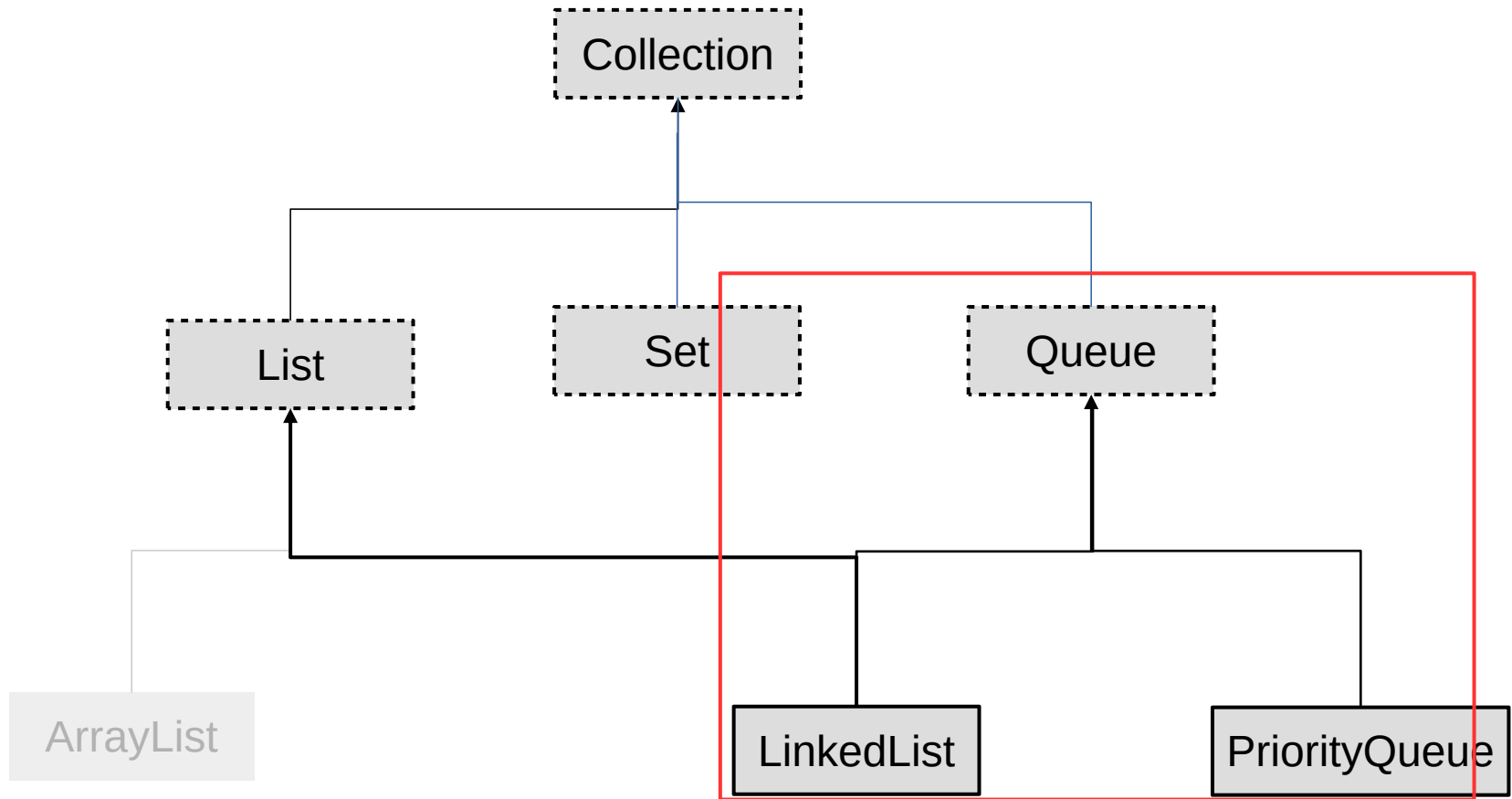
```
import java.util.*

public class SetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Set<Integer> intset = new HashSet<Integer>();
        for (int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
}
```

Set

- TreeSet
 - 特点: 速度较慢(插入, 删除, 查找), 有序
- LinkedHashSet
 - 特点: 速度快, 按插入顺序排列

Queue



Queue

- Queue 接口
 - 队列规则: 先进先出 (First in, First out)
 - 接口
 - offer(Object o), add(Object o): 将对象加入队列尾部
 - poll(), remove(): 弹出位于队首的对象
 - peek(), element(): 返回位于队首的对象, 并不删除
 - LinkedList
 - PriorityQueue


```
import java.util.*
```

```
public class QueueTest {  
    public static void printQ(Queue queue) {  
        while(queue.peek() != null)  
            System.out.print(queue.remove() + " ");  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        Queue<Integer> queue = new LinkedList<Integer>();  
        Random rand = new Random(47);  
        for(int i = 0; i < 10; i++)  
            queue.offer(rand.nextInt(i + 10));  
        printQ(queue);  
  
        Queue<Character> qc = new LinkedList<Character>();  
        for(char c : "Brontosaurus".toCharArray())  
            qc.offer(c);  
        printQ(qc);  
    }  
}
```

Queue

- PriorityQueue

- 优先级队列

- 每次出队时, 选择优先级最高的对象
 - 队列中的对象可以比较优先级
 - 普通队列也可看成优先级队列: 优先级为加入队列的时间

```
import java.util.*;
public class PriorityQueueTest {
    public static void main(String[] args) {
        PriorityQueue<Integer> qi = new PriorityQueue<Integer>();
        int [ ] iarray = {25, 22, 20, 18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25};
        for (int i: iarray)
            qi.offer(i);
        printQ(qi);

        PriorityQueue<Character> qc = new PriorityQueue<Character>();
        for(char c : "Brontosaurus".toCharArray())
            qc.offer(c);
        printQ(qc);
    }
}
```

Queue

- 自定义优先级

- 构造函数

```
PriorityQueue<E>(int initialCapacity, Comparator<E> comparator)
```

- Comparator 接口

- 定义两个元素的优先级关系

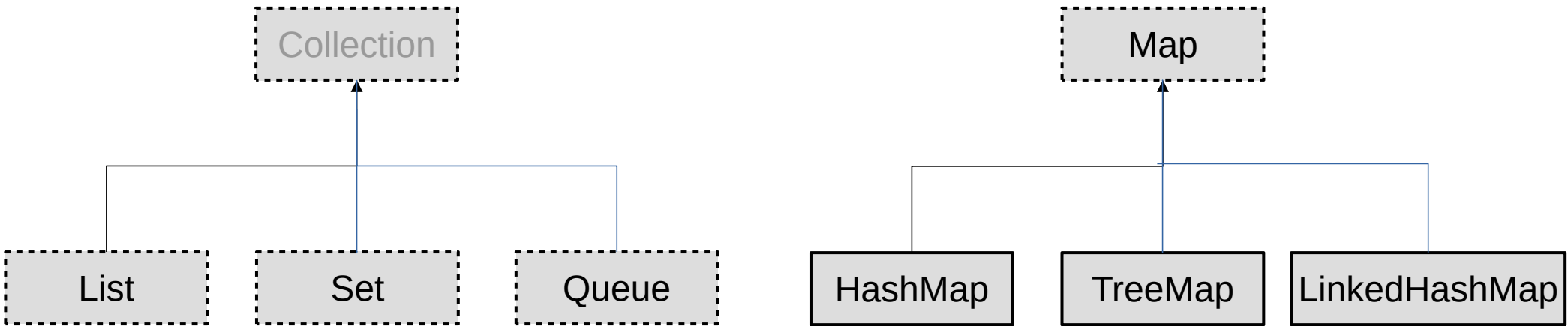
- 包含方法 `compare(E e1, E e2)`

- Compare 返回负数当 e1 优先级低于 e2
 - Compare 返回正数当 e1 优先级高于 e2
 - Compare 返回0当 e1 优先级等于 e2

```
public class PriorityQueueTest {  
    public static void main(String[] args) {  
        PriorityQueue<Character> qc = new PriorityQueue<Character>();  
        for(char c : "Brontosaurus".toCharArray())  
            qc.offer(c);  
        printQ(qc);  
    }  
}
```

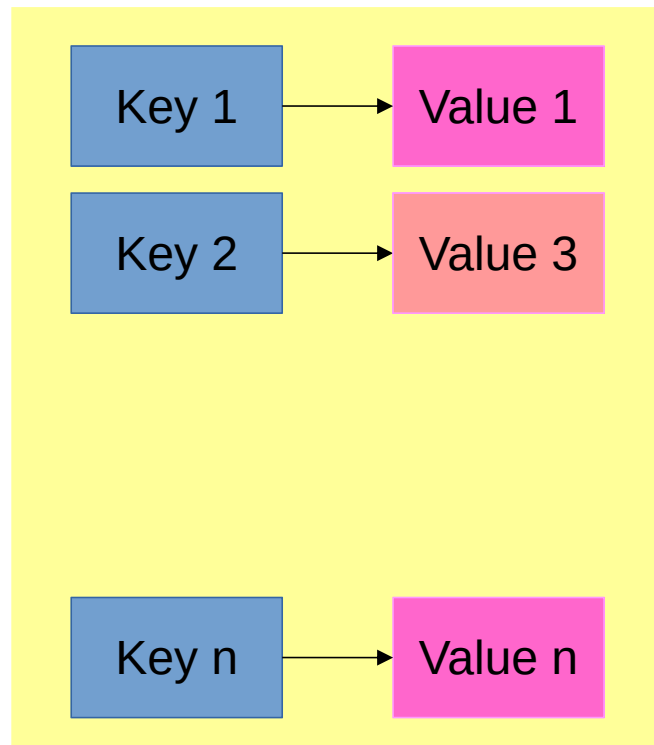
```
        PriorityQueue<Character> rqc = new PriorityQueue<Character>(10,  
            new Comparator<Character>(){  
                public int compare(Character c1, Character c2){  
                    if (c1 > c2)  
                        return -1;  
                    else if (c1 < c2)  
                        return 1;  
                    else  
                        return 0;  
                }  
            });  
        for(char c : "Brontosaurus".toCharArray())  
            rqc.offer(c);  
        printQ(rqc);  
    }  
}
```

Map



Map

- Map
 - Key-value pair



Map

- Map 接口
 - 存入键值对: `put(K key, V value)`
 - 返回键对应的值: `get(K key)`
 - 是否包含键`key`: `containsKey(Object key)`
 - 是否包含值`value`: `containsValue(Object value)`
 - 返回键组成的Set: `keySet()`
 - 返回值组成的Collection: `values()`


```
import java.util.*
```

```
public class MapTest {  
    public static void main(String[] args) {  
        HashMap<String, String>m = new HashMap<String, String>()  
        m.put("rat", "Fuzzy");  
        m.put("cat", "Rags");  
        m.put("dog", "Bosco");  
        m.put("dog", "Spot");  
        System.out.println(m.get("dog"));  
        System.out.println(m.containsKey("dog"));  
        System.out.println(m.containsValue("dog"));  
        System.out.println(m.keySet());  
        System.out.println(m.values());  
    }  
}
```

```

import java.util.*

public class MapTest {
    public static void main(String[] args) {
        HashMap<String, ArrayList<Integer>>m = new HashMap<>()
        m.put("rat", new ArrayList<Integer>(Array.asList(1,2,3)));
        m.put("cat", new ArrayList<Integer>(Array.asList(4,5,6)));
        m.put("dog", new ArrayList<Integer>(Array.asList(7,8,9)));
        m.put("dog", new ArrayList<Integer>(Array.asList(10,11,12)));
        System.out.println(m.get("dog"));
        System.out.println(m.containsKey("dog"));
        System.out.println(m.keySet());
        System.out.println(m.values());

        HashMap<String, HashMap<String, Integer>>n = new HashMap<>();
        ...
    }

```

Array.asList:

- 将数组转化成List ([文档](#))
- 包: java.util.Array
- 可变参数数量

Map

- Map
 - 建议: Immutable key

迭代器与foreach

- Iterable 接口
 - 提供iterator(): 返回迭代器
- Collection扩展了Iterable接口
- foreach 语句
 - 对所有实现Iterable 接口的类
 - 数组

