

华中科技大学

课程实验报告

课程名称： Java 语言程序设计

实验名称： 泛型栈模拟泛型队列

院 系： 计算机科学与技术

专业班级： 信息安全 1503 班

学 号： U201514858

姓 名： 陈 薇

指导教师： 吕新桥

2018 年 05 月 22 日

一、需求分析

1. 题目要求

参见 <https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>, Java 提供的 `java.util.Queue` 是一个接口没有构造函数。试用 `java.util.Stack<E>` 泛型栈作为父类, 用另一个泛型栈对象作为成员变量, 模拟实现一个泛型子类 `Queue<E>`, 当存储元素的第一个栈的元素超过 `dump` 时, 再有元素入队列就倒入第 2 栈。除提供无参构造函数 `Queue()` 外, 其它所有队列函数严格参照 `java.util.Queue` 的接口定义实现。

```
import java.util.Stack;
import java.util.NoSuchElementException;
public class Queue<E> extends Stack<E>{
    public final int dump=10;
    private Stack<E> stk;
    public Queue(){ /* 在此插入代码*/ }
    public boolean add(E e) throws IllegalStateException, ClassCastException,
        NullPointerException, IllegalArgumentException{ /* 在此插入代码*/ }
    public boolean offer(E e) throws ClassCastException, NullPointerException,
        IllegalArgumentException{ /* 在此插入代码*/ }
    public E remove() throws NoSuchElementException { /* 在此插入代码*/ }
    public E poll() { /* 在此插入代码*/ }
    public E peek() { /* 在此插入代码*/ }
    public E element() throws NoSuchElementException { /* 在此插入代码*/ }
}
```

考虑到 `Stack<E>` 只能存储类型为 `E` 的元素, 以及 `Stack` 是一个存储能力 (capacity, 参见有关说明) 理论上无限的类型, 这可能会影响到相关方法的异常处理, 请适当处理上述异常 (也许某些异常从来都不会发生)。

思考: `Queue<E>` 是否应该提供 `clone` 和 `equals` 函数, 以及其它一些函数如 `addAll` 等?

2. 需求分析

参照 <https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html> 文档内容和题目需求, 总结 `Queue<E>` 类中的属性含义和方法功能如下。

1、属性: 本类中包含两个属性

```
public final int dump = 10;
private Stack<E> stk;
```

该实验利用两个栈模拟先进先出的队列, `Queue` 类继承于栈 `Stack` 类, 从父类中继承到一个栈, 用作压栈, 另一个用作出栈的栈以成员变量的形式包含在类 `Queue` 中, 即为属性 `stk`。

常量属性 **dump** 用来限制该队列的大小，当存储元素的第一个栈的元素超过 **dump** 时，再有元素入队列就倒入第 2 栈。因此本实验实现队列的容量为 $\text{dump}+1\sim 2*\text{dump}$ 。

2、方法

本类中的方法基于队列先进先出的原则，实现了对队列进行的一系列操作，例如出队列、入队列、克隆等。

(1) **QUEUE();**

类 **QUEUE** 的无参构造函数，在该函数内调用 **Stack<E>** 的构造函数对引用变量 **stk** 进行初始化。

(2) **boolean add(E e)**

该函数在队列未满的情况下将特定类型的元素 **e** 插入到队列的尾部，若插入成功则返回 **true**，否则抛出异常。异常有以下四种类型：

- a. **IllegalStateException**-若队列已满，则抛出该异常
- b. **ClassCastException**-若元素 **e** 的类型和队列规定类型不符，则抛出该异常
- c. **NullPointerException**-若元素 **e** 为 **null** 且该队列不允许插入 **null**，则抛出该异常
- d. **IllegalArgumentException**-若元素 **e** 的某些属性不允许它加入队列中，则抛出该异常

(3) **E remove()**

该函数检索并删除队列的首元素，若队列为空则抛出 **NoSuchElementException** 类型的异常，反之则删除首元素并返回首元素的值。

(4) **E element()**

该函数检索但不删除队列首元素，若队列为空则抛出 **NoSuchElementException** 类型的异常，反之则返回首元素的值。

(5) **boolean offer(E e)**

该函数在队列未满的情况下将特定类型的元素 **e** 插入到队列的尾部，若插入成功则返回 **true**，否则返回 **false**。该函数也会抛出异常，异常有以下三种类型：

- a. **ClassCastException**-若元素 **e** 的类型和队列规定类型不符，则抛出该异常
- b. **NullPointerException**-若元素 **e** 为 **null** 且该队列不允许插入 **null**，则抛出该异常
- c. **IllegalArgumentException**-若元素 **e** 的某些属性不允许它加入队列中，则抛出该异常

(6) **E poll()**

该函数检索并删除队列的首元素，若队列为空则返回 **null**，否则删除首元素并返回首元素的值。

(7) **E peek()**

该函数检索但不删除队列首元素，若队列为空则返回 **null**，否则返回首元素的值。

(8) **boolean addAll(Collection<? extends E> c)**

该函数将 **c** 中的所有元素一次性添加到队列中，若 **c** 为 **null**，则抛出 **NullPointerException** 类型的异常，若添加 **c** 中元素的过程中队列已满，则返回 **false**，否则返回 **true**。

(9) **boolean equals(Object o)**

该函数覆盖父类 `equal` 函数，判断对象 `o` 和当前对象是否相等，若 `o` 为 `Queue` 类型的对象且两者相等则返回 `true`，否则返回 `false`。

(10) `Object clone()`

该函数覆盖父类的 `clone` 函数，将当前对象 `clone` 成为一新的对象，并作为返回值返回。

由此可以看出，以上 2-7 这六个函数可以分为两类，2-4 这三个函数在操作失败时抛出异常，另外三个函数则在失败时返回一个特殊的值（`null` 或 `false`）。

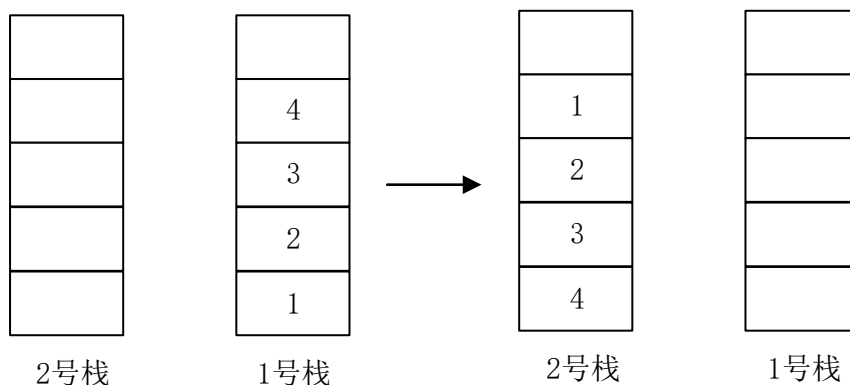
二、系统设计

1. 概要设计

栈的特征为先进后出，而队列的特征为先进先出，因此若使用两个栈模拟队列，基本思路为一个栈用于入队列，一个栈用于出队列，假定 1 号栈用于入队列，2 号栈用于出队列。

对于入队列，直接将元素压入 1 号栈。而对于出队列，为保证队列先进先出的顺序，需要改变首元素在栈中的位置，使其由栈底变化至栈顶，此时需要借助 2 号栈。首先判断 2 号栈是否有元素，若有则直接将 2 号栈的栈顶弹出，若 2 号栈无元素，则依次将 1 号栈元素弹出后压入 2 号栈，然后弹出栈顶作为队首，举例如下。

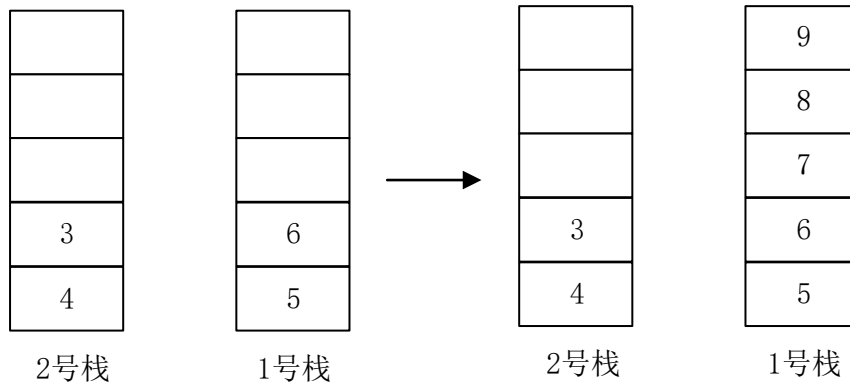
假定使用两个大小为 5 的栈模拟一大小为 10 的队列。首先将 1、2、3、4 四个元素依次入队列，则直接将 4 个元素压入 1 号栈。其次将两个元素出队列，为保证先进先出，将 1 号栈的元素依次出栈，然后压入 2 号栈后弹出 1 和 2，弹出后队列中包含两个元素 3 和 4。



此时，要求将 5、6、7、8、9 五个元素入队列，直接将五个元素压入 1 号栈，若仍要求元素入队列的话，尽管 2 号栈中有剩余空间，但若采用上述做法则会破坏队列先进先出的原则，因此此时队列已满。

在下图所示情况下执行出队列操作，由于 2 号栈已有元素，可直接将 2 号栈的首元素 3 弹出，即为队首。

综上所述，对于所有对队列进行的操作，2 号栈有元素和 2 号栈无元素采用两种不同的处理方式。且队列是否已满的判断条件并非 1 号栈和 2 号栈均已满，因此需分情况讨论，具体每个操作的思想如下。



(1) 入队列

若 1 号栈未满，则将元素压入 1 号栈；

若 1 号栈已满且 2 号栈为空，则将 1 号栈的元素依次出栈然后压入 2 号栈，再将元素压入已空的 1 号栈；

若 1 号栈已满但 2 号栈有元素，则此时该队列已满，执行错误。

(2) 出队列

若 2 号栈为空且 1 号栈为空，则队列为空，执行错误。

若 2 号栈为空，则将 1 号栈的元素依次出栈然后压入 2 号栈，再从 2 号栈弹出栈首元素，即为队列队首；

若 2 号栈不为空，则直接弹出 2 号栈栈首元素为队列队首；

(3) 查看队首元素

若 2 号栈为空且 1 号栈为空，则队列为空，执行错误。

若 2 号栈不为空，则将 1 号栈的元素依次出栈然后压入 2 号栈，再从 2 号栈获取栈首元素，即为队列队首；

若 2 号栈不为空，则直接从 2 号栈获取栈首元素为队首元素；

2. 详细设计

本部分主要介绍 Queue 类中各方法的处理流程、功能、入口参数、返回值等信息。

1、构造函数 QUEUE();

参数：无参数；

返回值：该函数为类 QUEUE 的构造函数，不存在返回值；

访问权限：public 公有函数

功能：初始化 Stack 引用类型的成员变量 stk；

流程：由于父类 STACK 中存在无参的构造函数，因此无需显式的调用父类的构造函数，直接利用 new 初始化属性 stk 即可。

2、boolean add(E e) throws IllegalStateException, ClassCastException, NullPointerException, IllegalArgumentException

参数：要入队列的元素 e；

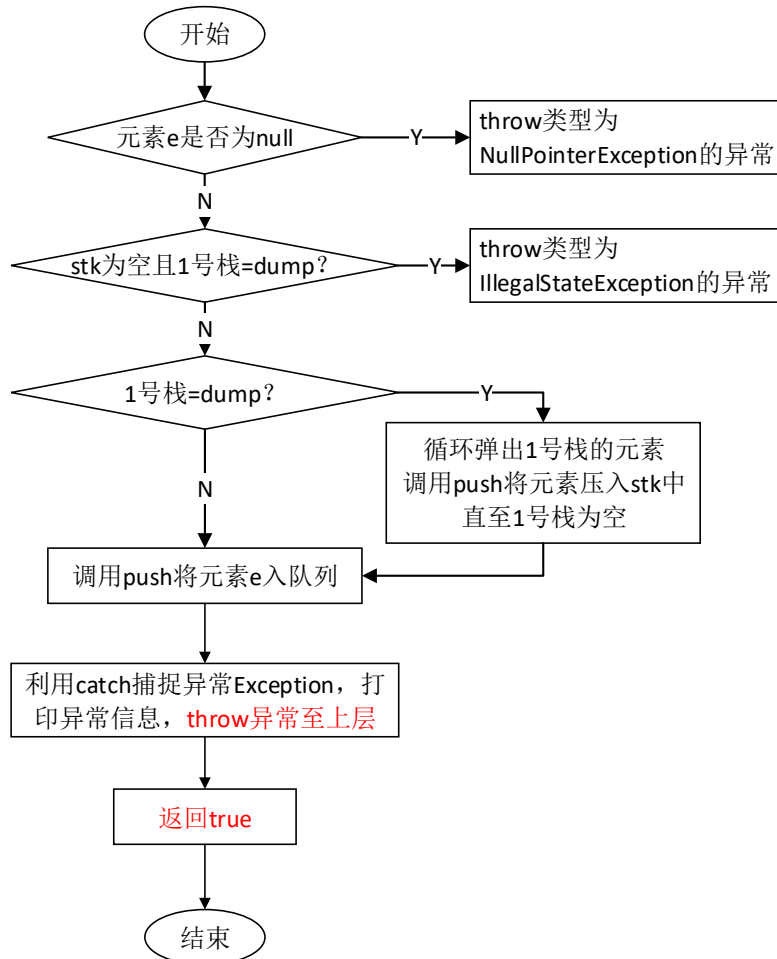
返回值：操作结果，入队列成功则返回 true，否则抛出异常；

访问权限: public

功能: 将元素 e 入队列;

流程图: 函数流程图如下图 2.3, 若操作从父类 stack 继承来的栈, 可以直接通过函数名调用相应函数, 若操作引用类型的 stk 栈, 则通过 stk.函数名的形式调用函数。

由于函数会抛出异常, 为打印异常信息, 利用 try...catch 语句捕捉异常, 在 try 体内完成正常处理流程, catch 体内打印异常信息后再利用 throw 向上层抛出。



在 try 体内, 若 e 为 null, 则抛出 NullPointerException 类型的异常, 若队列已满, 则抛出 IllegalStateException 的异常。若均正常, 则插入元素。

在 catch 体内, 捕获所有的 Exception 异常, 打印错误信息后, 不处理异常, 直接向上层传递, 因此函数声明时需要使用 throws 声明未处理的异常。

若有异常, 则在 catch 体内执行 throw 之后函数流程就已结束, 只有在没有任何异常的情况下, 才会执行 catch 后面的语句, 返回 true 的语句, 此时说明插入成功。

3、boolean offer(E e) throws ClassCastException, NullPointerException, IllegalArgumentException

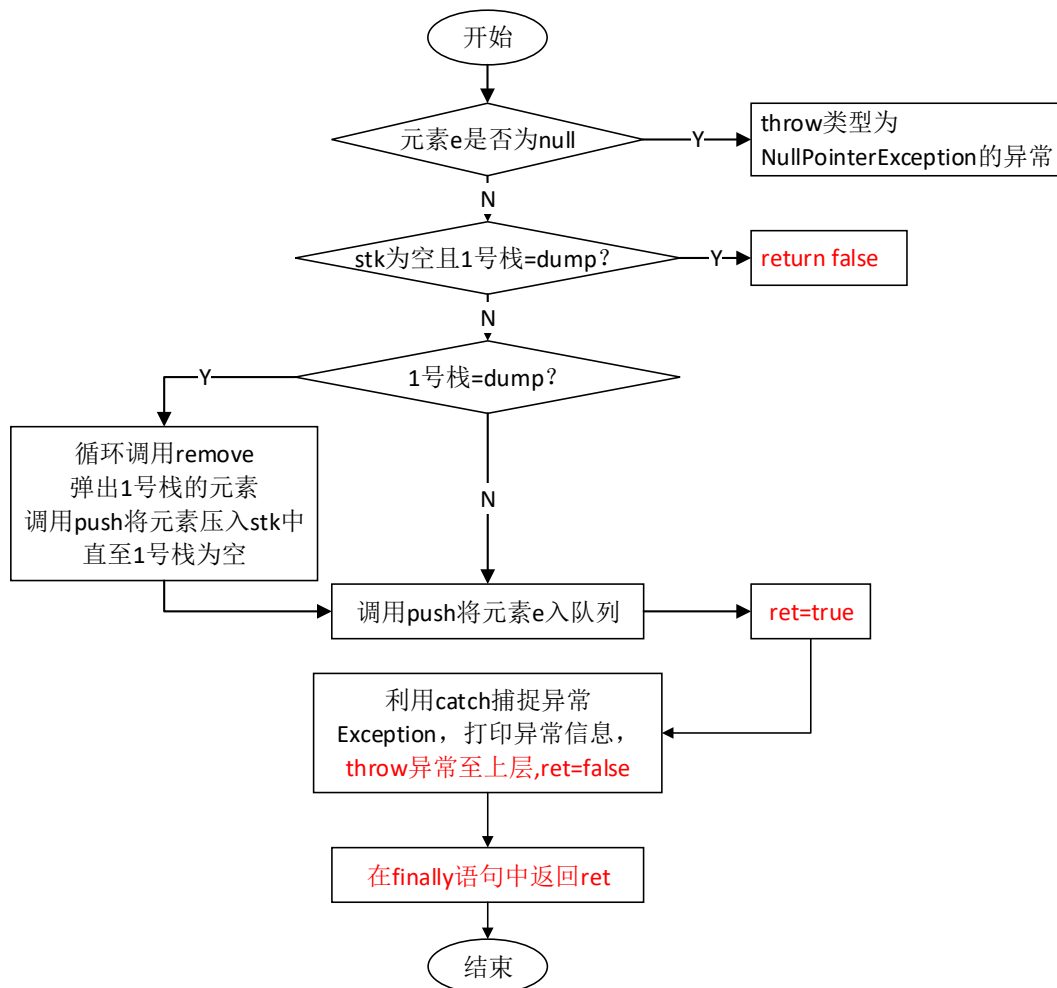
参数: 要入队列的元素 e;

返回值: 操作结果, 入队列成功则返回 true, 否则返回 false;

访问权限: public

功能：将元素 **e** 入队列；

流程图：函数流程图如下图，该函数流程与 **add** 函数基本类似，只是在队列已满时直接返回 **false**，不抛出异常。同样为打印异常信息，利用 **try...catch** 语句捕捉异常，在 **catch** 语句体内打印异常信息后再利用 **throw** 向上层抛出。另外，为达到操作成功返回 **true**，操作失败返回 **false** 的效果，在 **try...catch** 语句后添加 **finally**，不论是否有异常，异常是否被捕获或处理，都会执行 **finally** 语句。因此设置布尔型的变量 **ret**，若执行成功则置 **ret** 为 **true**，否则在 **catch** 语句体内设置为 **false**，最后在 **finally** 体内返回 **ret**。



4、E remove() throws NoSuchElementException

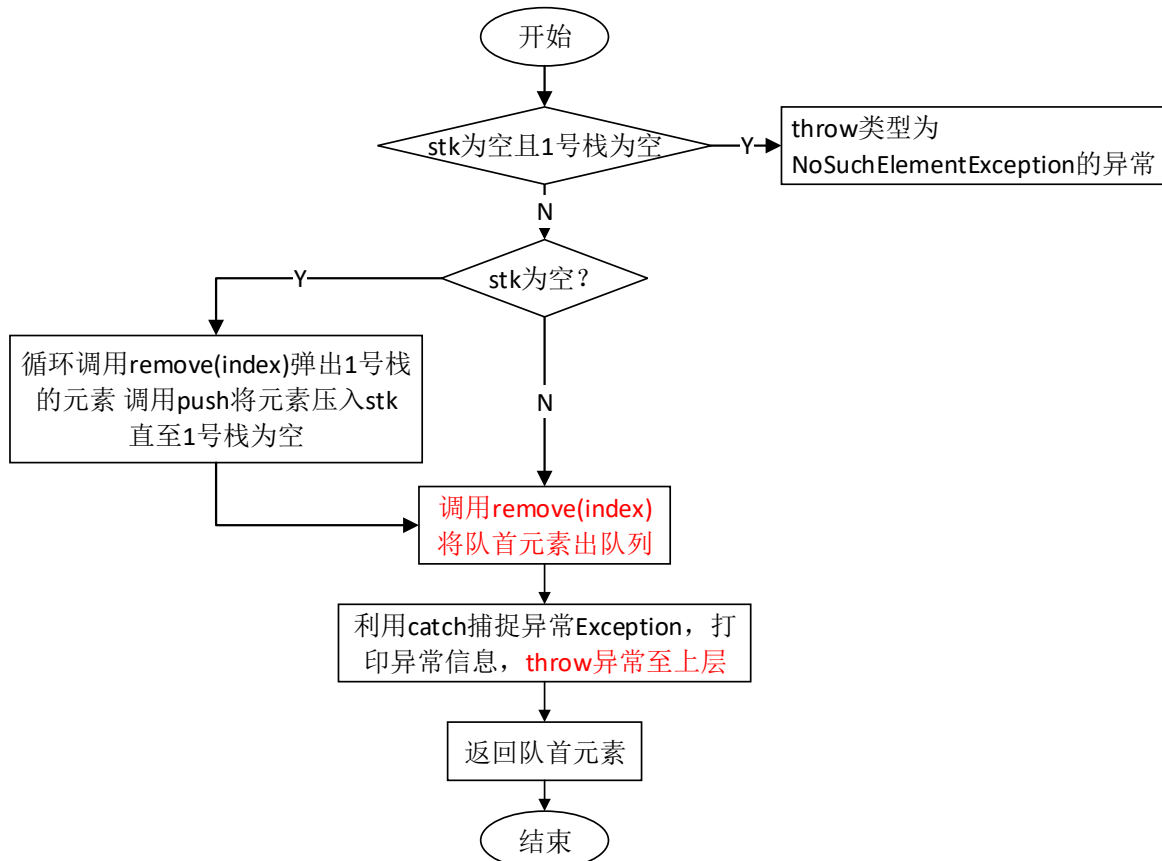
参数：无参数；

返回值：若队列不为空，则返回类型为 **E** 的队列首元素，反之抛出异常；

访问权限：public

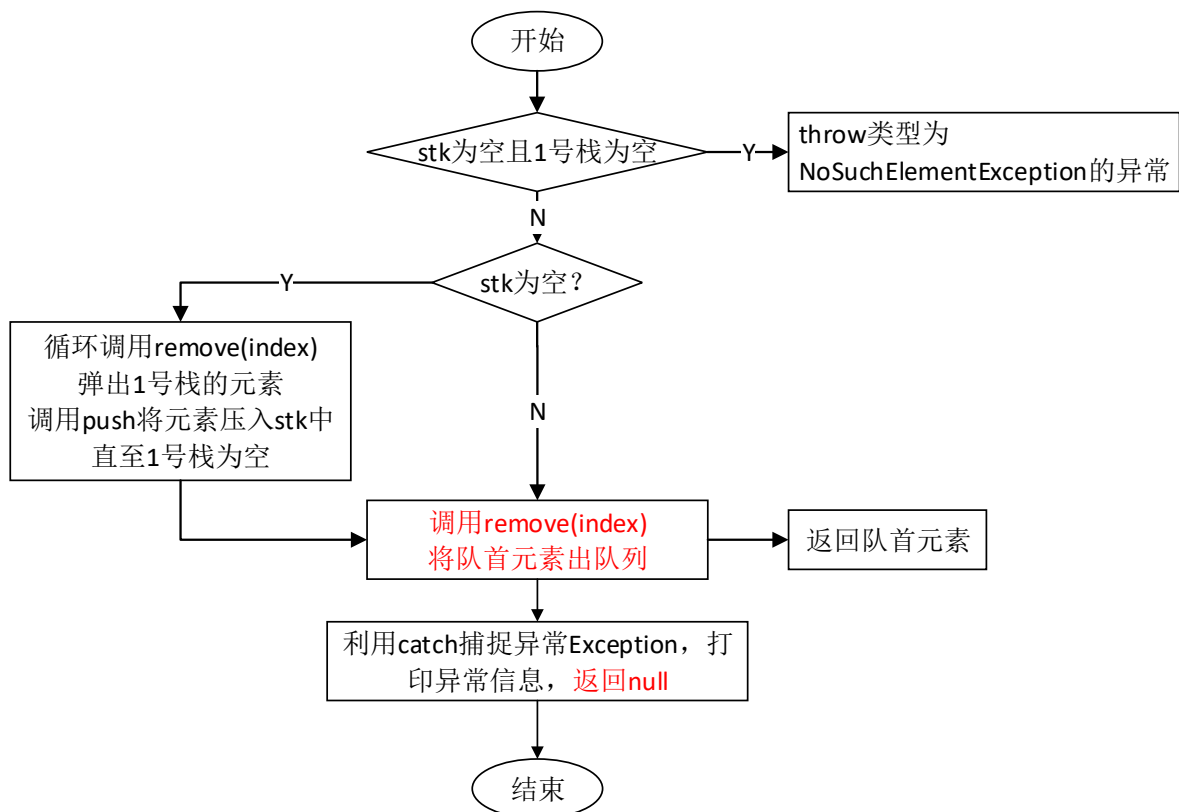
功能：将队列首元素出队列并作为返回值返回；

流程图：函数流程图如下图，利用 **try...catch** 语句，在 **try** 语句体中，若栈为空则抛出 **NoSuchElementException** 的异常，否则执行正常的出队列逻辑，在 **catch** 体内捕获异常，打印异常信息，并向上层抛出异常。最后无异常情况下，返回队首元素。



5、E poll()

参数：无参数；



访问权限: **public**

返回值: 若队列不为空, 则返回类型为 **E** 的队列首元素, 反之返回 **null**;

功能: 将队列首元素出队列并作为返回值返回;

流程图: 函数流程图如下图, 该函数基本逻辑与 **remove** 类似, 只是该函数在 **catch** 到异常后并不继续向上层抛出, 而是直接返回 **null**。

6、E element() throws NoSuchElementException

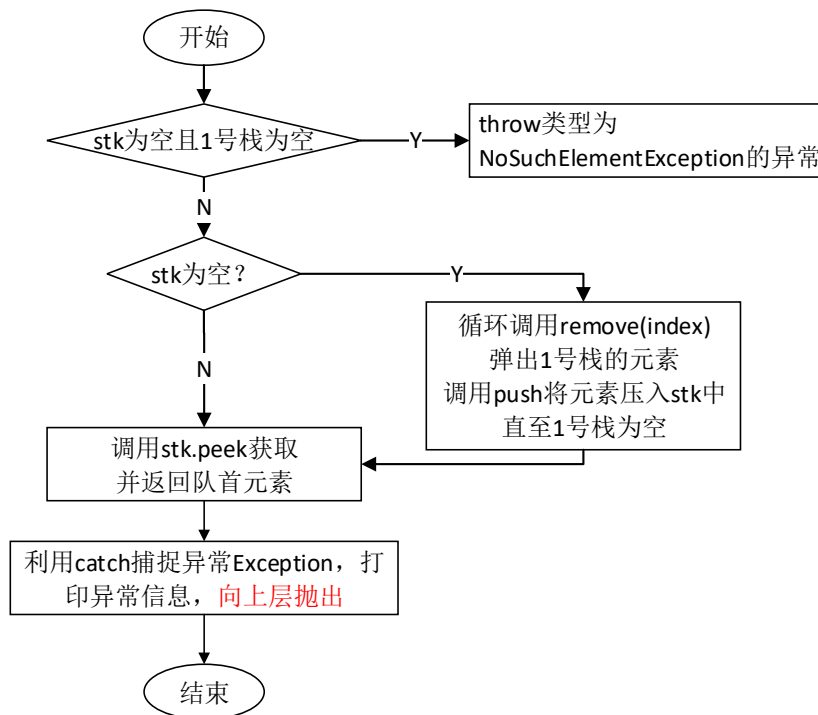
参数: 无参数;

返回值: 若队列不为空, 返回类型为 **E** 的队首元素, 否则抛出异常。

访问权限: **public**

功能: 获取队首元素;

函数流程: 该函数流程图如下。函数流程同样和 **remove** 类似, 不过只获取 **stk** 的栈顶元素 (队首元素) 而不将其弹出, 即不将其出队列。



7、E peek ()

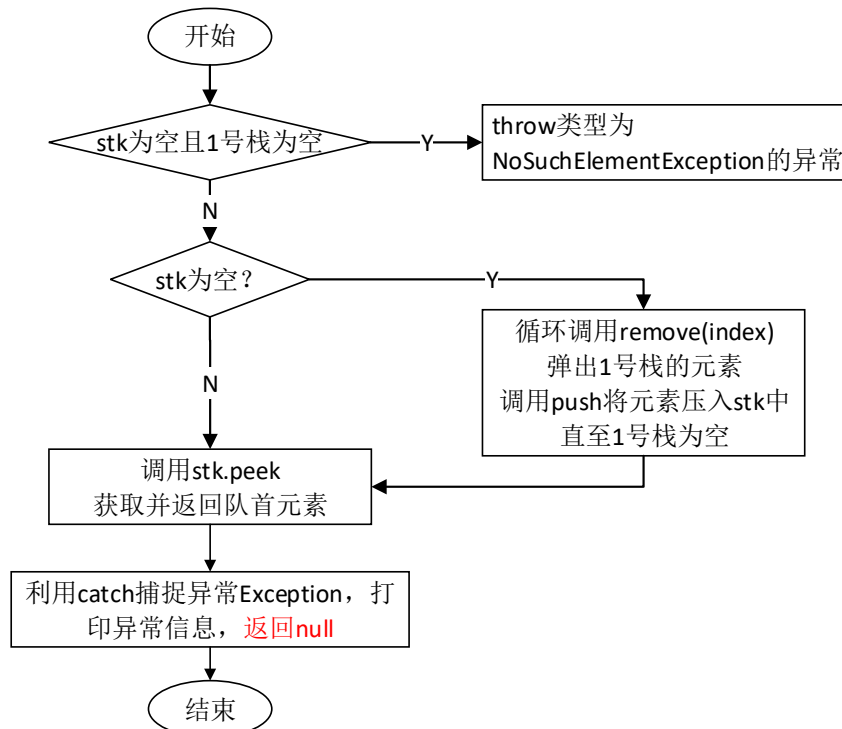
参数: 无参数;

返回值: 若队列不为空, 返回类型为 **E** 的队首元素, 否则返回 **null**。

访问权限: **public**

功能: 获取队首元素;

函数流程: 该函数流程图如下。函数流程和 **poll** 类似, 不过在捕获异常后, 并不继续传递至上层, 而是直接返回 **null**。



8、synchronized boolean addAll(Collection<? extends E> c) throws

NullPointerException

参数：要添加至队列的元素的集合 **c**；

返回值：若全部添加成功，返回 **true**，否则返回 **false**。

访问权限：public

功能：将 **c** 中的所有元素添加至队列中；

函数流程：若 **c** 为 **null**，则抛出 **NullPointerException** 类型的异常。若不为 **null**，则对于 **c** 中的每一个元素，循环调用 **offer** 函数，并获得 **offer** 函数的返回值，若返回 **false** 说明插入时发生异常，**addall** 函数直接返回 **false**，否则所有元素均添加成功，返回 **true**。

9、synchronized boolean equals(Object o)

参数：进行比较的对象 **o**；

返回值：若当前对象和 **o** 相等，返回 **true**，否则返回 **false**。

访问权限：public

功能：比较当前对象和 **o** 是否相等；

函数流程：首先判断 **o** 是否为 **Queue** 类型的对象，若不是则直接返回 **false**。若是则调用 **super.equal** 和 **stk.equal**，分别比较继承的栈和 **stk** 栈是否都相等，两者均为 **true** 时，返回 **true**。

10、synchronized Object clone()

参数：无参数

返回值：返回克隆成功的对象

访问权限：public

功能：返回当前对象的克隆

函数流程：分别调用父类的 clone 和 stk.clone 实现两个栈的克隆，进而实现队列的克隆。

三、软件开发

开发环境：IntelliJ IDEA

JDK 版本：jdk_1.8.0_144

四、软件测试

1、测试 clone、equals、addAll 的功能，测试代码如下图所示。

```
Queue<Integer> integers = new Queue<>();
Queue<String> strings = new Queue<>();
Collection<Integer> intList = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
    12, 13, 14, 15, 16, 17, 18, 19, 20));
Collection<String> strList = new ArrayList<>(Arrays.asList("xxx", "yyy", "zzz"));
boolean int_ret = integers.addAll(intList);
boolean str_ret = strings.addAll(strList);
System.out.print("integers.equals(strings) is " + integers.equals(strings) + "\n");
Queue<Integer> others = (Queue<Integer>)integers.clone();
System.out.print("integers.equals(others) is " + integers.equals(others) + "\n");
System.out.print("integers == others is " + (integers == others) + "\n");
for (int i = 0; i < 20; i++) {
    Integer obj = integers.remove();
    System.out.print(obj + " ");
}
System.out.print("\n");
```

由于类 Queue 中含有引用型的变量，因此应设计 clone、equals 函数，并且为添加元素方便，提供 addAll 函数，以实现元素的一次性添加。

(1)测试 addAll 函数：代码首先分别创建 Integer 和 String 类型的队列 integers 和 strings，创建两个 List 即 intList 和 strList 存放将要一次性添加到队列中的数据。

```

v  integers = {Queue@527} size = 10
  > 0 = {Integer@532} 11
  > 1 = {Integer@533} 12
  > 2 = {Integer@534} 13
  > 3 = {Integer@535} 14
  > 4 = {Integer@536} 15
  > 5 = {Integer@537} 16
  > 6 = {Integer@538} 17
  > 7 = {Integer@539} 18
  > 8 = {Integer@540} 19
  > 9 = {Integer@541} 20
v  strings = {Queue@528} size = 3
  > 0 = "xxx"
  > 1 = "yyy"
  > 2 = "zzz"
```

分别调用 `integers.addAll` 和 `strings.addAll` 将 `list` 中的数据插入队列。添加完成后，调试运行该代码，两个队列中内容如上。调试中只显示从 `Stack` 继承来的元素的内容，由于 `stk` 不可见，因此对于添加了 20 个元素的队列 `integers`，未显示出另外 10 个元素。但可以看出，用于入栈的 1 号栈存放了后 10 个元素，可知前十个元素被放入 `stk` 栈中。

(2) 测试 `equals` 和 `clone` 函数

代码中测试 `integers` 和 `strings` 是否相等，显然两者不等，此时结果应为 `false`;

对于 `clone` 函数，应实现深拷贝赋值，截图 `clone` 后 `other` 引用变量和 `integers` 引用变量发现两者的 `hashCode` 并不一致，说明未引用同一对象，克隆成功。

```
> integers = {Queue@527} size = 10
> strings = {Queue@528} size = 3
> intList = {ArrayList@529} size = 20
> strList = {ArrayList@530} size = 3
  int_ret = true
  str_ret = true
> others = {Queue@550} size = 10

Initial Queue Class!
Initial Queue Class!
integers.equals(strings) is false
integers.equals(others) is true
integers == others is false
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

测试 `equals` 函数，在 `others` 和 `integers` 内容相等的情况下，看结果是否为 `true`，同时为对比，打印 `others == integers` 的内容，判断是否引用同一对象。可以看到 `integers` 和 `others` 并不引用同一对象，但两者内容相等，因此 `integers.equals(others)` 结果为 `true`。

同时为仔细看到 `integers` 的元素，利用 `remove` 循环将元素移除队列后打印元素的值，和添加的值和序列保持一致，`addAll` 函数执行成功。

2、测试 `remove`、`poll`、`peek` 和 `element` 函数的异常情况

```
try {
    System.out.print("integers.poll() is " + integers.poll() + "\n");
    System.out.print("integers.remove() is " + integers.remove() + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}
```

Queue > main()

```
Queue
E:\jdk1.8.0_144\bin\java ...
Initial Queue Class!
Initial Queue Class!
integers.equals(strings) is false
integers.equals(others) is true
integers == others is false
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
null
integers.poll() is null
null
java.util.NoSuchElementException
```

在上述代码执行完毕后，`integers` 对象无内容，测试出队列函数。

poll 函数在队列为空时并未抛出异常，而直接返回 null，remove 函数在队列为空时，抛出异常，导致语句未正常打印，catch 异常后打印异常信息类型，为 NoSuchElementException。

同样的，peek 函数在队列为空时也直接返回 null，而 element 函数抛出异常。

3、测试 add 和 offer 函数的正常情况

在上述代码执行完毕后，integers 对象无内容，测试 add 和 offer 函数，打印执行结果。

```

System.out.print("integers.add(1) is " + integers.add(1) + "\n");
System.out.print("integers.offer(2) is " + integers.offer(2) + "\n");
Queue > main()
Queue
E:\jdk1.8.0_144\bin\java ...
Initial Queue Class!
Initial Queue Class!
integers.equals(strings) is false
integers.equals(others) is true
integers == others is false
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
integers.add(1) is true
integers.offer(2) is true

```

元素添加成功，结果均为 true。此时若再利用 remove、poll、peek 和 element 函数，则结果正确。

```

try {
    System.out.print("integers.peek() is " + integers.peek() + "\n");
    System.out.print("integers.element() is " + integers.element() + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}
Queue > main()
Queue
E:\jdk1.8.0_144\bin\java ...
Initial Queue Class!
Initial Queue Class!
integers.equals(strings) is false
integers.equals(others) is true
integers == others is false
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
null
integers.poll() is null
null
java.util.NoSuchElementException
null
integers.peek() is null
null
java.util.NoSuchElementException

```

```

System.out.print("integers.add(1) is " + integers.add(1) + "\n");
System.out.print("integers.offer(2) is " + integers.offer(2) + "\n");
try {
    System.out.print("integers.peek() is " + integers.peek() + "\n");
    System.out.print("integers.element() is " + integers.element() + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}

try {
    System.out.print("integers.poll() is " + integers.poll() + "\n");
    System.out.print("integers.remove() is " + integers.remove() + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}
}

Queue > main()
Queue
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
integers.add(1) is true
integers.offer(2) is true
integers.peek() is 1
integers.element() is 1
integers.poll() is 1
integers.remove() is 2

```

peek 函数和 element 函数均只获取首元素而不出队列，因此执行结果均为 1，poll 和 remove 函数将元素出队列，因此结果依次为 1、2。

4、测试 add 和 offer 函数的异常情况

再次将整型队列添加元素至队列已满，此时调用 add 和 offer 函数，结果如下。

```

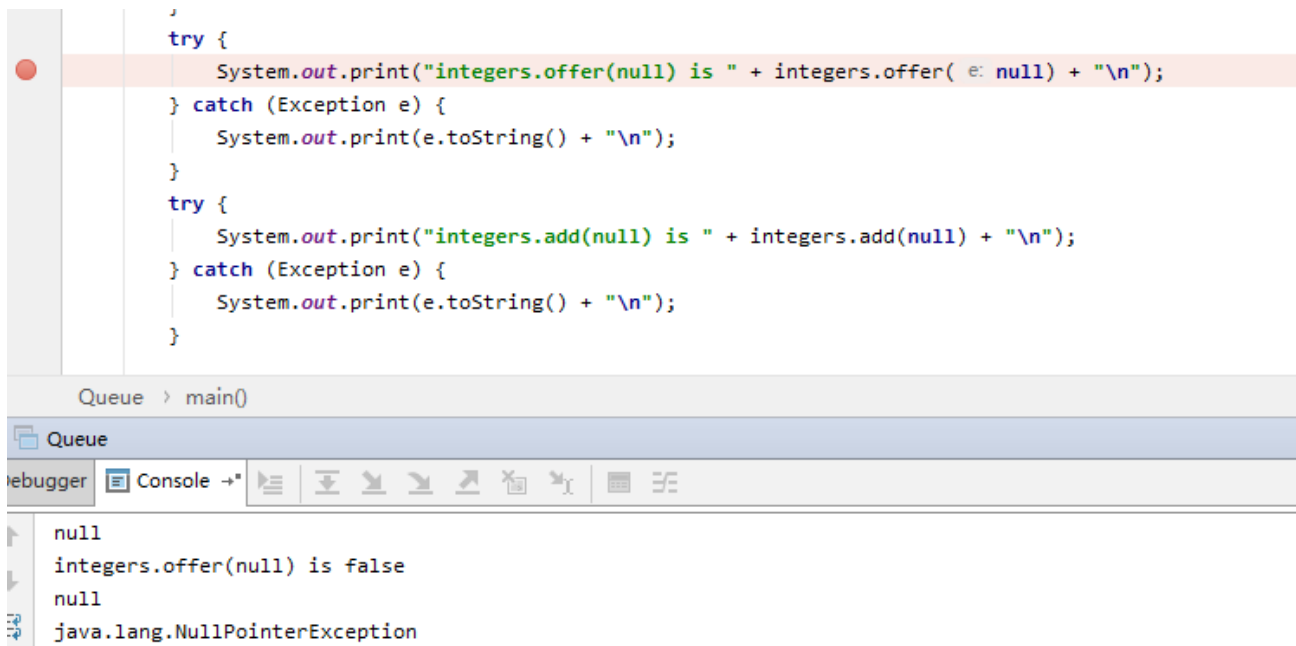
integers.addAll(intList);
try {
    System.out.print("integers.offer(5) is " + integers.offer(5) + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}

try {
    System.out.print("integers.add(5) is " + integers.add(5) + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}
}

Queue > main()
Queue
integers.offer(5) is false
null
java.lang.IllegalStateException

```

由于队列已满, `offer` 函数返回 `false`, 而 `add` 函数抛出 `IllegalStateException` 类型的异常。另外测试添加空元素的情况如下。



```
try {
    System.out.print("integers.offer(null) is " + integers.offer( e: null) + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}
try {
    System.out.print("integers.add(null) is " + integers.add(null) + "\n");
} catch (Exception e) {
    System.out.print(e.toString() + "\n");
}
```

Queue > main()

Queue

Debugger Console

```
null
integers.offer(null) is false
null
java.lang.NullPointerException
```

`offer` 函数返回 `false` 而 `add` 函数抛出 `NullPointerException` 类型的异常。

由于当类型不正常时编译会报错, 因此无法测试 `ClassCastException` 和 `IllegalArgumentException` 类型的异常, 也许该类型异常永远不会抛出。

五、特点与不足

1. 技术特点

设计了 `addAll` 函数、`clone` 函数实现深拷贝和 `equals` 函数判断两个队列的内容是否相等, 并且对于基本的六个函数, 有异常时均捕捉异常并打印了异常信息的内容。

2. 不足和改进的建议

对于异常信息的处理不是很规范, 重载 `AddAll`、`clone`、`equals` 函数时也未捕捉常见的一场情况, 并且仅测试了编写代码的基本功能。

六、过程和体会

1. 遇到的主要问题和解决方法

比较难处理的是, 起初打算利用从父类继承来的 `pop` 函数实现栈顶元素的弹出, 但查看父类 `Stack` 的源码发现, `pop` 函数会调用 `peek` 函数获取首元素的值, 而 `peek` 函数在 `Queue` 类中被覆盖, 且 `peek` 中在 `stk` 为空时仍然用到了 `pop` 函数, 最后导致 `pop` 和 `peek` 函数反复调用, 导致死循环。


```
/**
 * Removes the object at the top of this stack and returns that
 * object as the value of this function.
 *
 * @return The object at the top of this stack (the last item
 *         of the Vector object).
 * @throws EmptyStackException if this stack is empty.
 */
public synchronized E pop() {
    E    obj;
    int   len = size();

    obj = peek();
    removeElementAt( index: len - 1);

    return obj;
}
```

查看父类 `stack` 的成员函数，为防止进入死循环，选择利用 `remove(size() - 1)` 实现类似 `pop` 的操作，其余函数中只要需要使用弹出栈顶元素，均使用 `remove(index)` 实现。


2. 课程设计的体会

通过本次实验，了解到 `java` 异常的基本知识和泛型编程的基本方法，收获颇深。

七、源码和说明

1. 文件清单及其功能说明

主要文件在 `javaexp1` 文件夹中，目录如下，该文件包含 `Queue` 类的实现和测试。

此电脑 > 文档 (F:) > javaexp > javaexp1 > Queue > src				
名称	修改日期	类型	大小	
 Queue.java	2018/5/22 星期...	Java 源文件	8 KB	

2. 用户使用说明书

利用 `IDEA` 打开 `Queue` 文件夹，点击运行即可查看命令行运行结果。同样也可利用 `javac` 生成 `class` 文件，再利用 `java` 运行该文件，如下图。


```
F:\javaexp\javaexp1\Queue\src>E:\jdk1.8.0_144\bin\javac.exe Queue.java
```

注：Queue.java使用了未经检查或不安全的操作。

注：有关详细信息，请使用 -Xlint:unchecked 重新编译。

```
F:\javaexp\javaexp1\Queue\src>E:\jdk1.8.0_144\bin\java.exe Queue
```

```
Initial Queue Class!
```

```
Initial Queue Class!
```

```
integers.equals(strings) is false
```

```
integers.equals(others) is true
```

```
integers == others is false
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
integers.add(1) is true
```

```
integers.offer(2) is true
```

```
integers.peek() is 1
```

```
integers.element() is 1
```

3. 源代码

Queue.java 的内容如下：

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

```
public class Queue<E> extends Stack<E> {
```

```
    public final int dump = 10;
```

```
    private Stack<E> stk;
```

```
    public Queue() {
```

```
        stk = new Stack<E>();
```

```
        System.out.println("Initial Queue Class!");
```

```
    }
```

```
    public boolean add(E e) throws IllegalStateException, ClassCastException,
        NullPointerException, IllegalArgumentException {
```

```
        try {
```

```
            if (e == null) {
```

```
                throw new NullPointerException();
```

```
            }
```

```
            if ((size() == dump) && !stk.empty()) {
```

```
                throw new IllegalStateException();
```

```
            }
```

```
            if (size() == dump) {
```

```
                E obj;
```

```
                while (!empty()) {
```

```
                    obj = remove(size() - 1);
```

```
                    stk.push(obj);
```

```
                }
```

```
            }
```

```
        push(e);
```

```

    } catch (Exception excp) {
        System.out.println(excp.getMessage());
        throw excp;
    }
    return true;
}

public boolean offer(E e) throws ClassCastException, NullPointerException,
    IllegalArgumentException {
    boolean ret = false;
    try {
        if (e == null) { //NullPointerException
            throw new NullPointerException();
        }
        if ((size() == dump) && !stk.empty()) {
            return false;
        }
        if (size() == dump) {
            E obj;
            while (!empty()) {
                obj = remove(size() - 1);
                stk.push(obj);
            }
        }
        push(e);
        ret = true;
    } catch (Exception excp) {
        System.out.print(excp.getMessage() + "\n");
        ret = false;
        throw excp;
    } finally {
        return ret;
    }
}

public E remove() throws NoSuchElementException {
    E obj;
    try {
        if (stk.empty() && empty()) {
            throw new NoSuchElementException();
        }
        if (stk.empty()) {
            while (!empty()) {
                obj = remove(size() - 1);
                stk.push(obj);
            }
        }
    }
}

```

```

        }
    }
    obj = stk.remove(stk.size() - 1);
} catch (NoSuchElementException excp) {
    System.out.println(excp.getMessage());
    throw excp;
}
return obj;
}

public E poll() {
    try {
        if (stk.empty() && empty()) {
            throw new NoSuchElementException();
        }
        E obj;
        if (stk.empty()) {
            while (!empty()) {
                obj = remove(size() - 1);
                stk.push(obj);
            }
        }
        obj = stk.remove(stk.size() - 1);
        return obj;
    } catch (Exception excp) {
        System.out.println(excp.getMessage());
        return null;
    }
}

public E element() throws NoSuchElementException {
    try {
        if (stk.empty() && empty()) {
            throw new NoSuchElementException();
        }
        E obj;
        if (stk.empty()) {
            while (!empty()) {
                obj = remove(size() - 1);
                stk.push(obj);
            }
        }
        obj = stk.peek();
        return obj;
    } catch (Exception excp) {

```

```

        System.out.println(excp.getMessage());
        throw excp;
    }
}

```

```

public E peek() {
    try {
        if (stk.empty() && empty()) {
            throw new NoSuchElementException();
        }
        E obj;
        if (stk.empty()) {
            while (!empty()) {
                obj = super.remove(size() - 1);
                stk.push(obj);
            }
        }
        obj = stk.peek();
        return obj;
    } catch (Exception excp) {
        System.out.println(excp.getMessage());
        return null;
    }
}

```

@Override

```

public synchronized boolean addAll(Collection<? extends E> c) throws
NullPointerException {
    boolean ret = true;
    if (c == null) {
        throw new NullPointerException();
    }
    Object[] a = c.toArray();
    for (Object e : a) {
        ret = offer((E) e);
        if (!ret) {
            return ret;
        }
    }
    return true;
}

```

@Override

```

public synchronized boolean equals(Object o) {
    if (o instanceof Queue) {

```

```

        return super.equals(o) && stk.equals(((Queue) o).stk);
    }
    return false;
}

@Override
public synchronized Object clone() {
    Queue<E> q = (Queue<E>) super.clone();
    q.stk = (Stack<E>) stk.clone();
    return q;
}

public static void main(String[] args) {
    Queue<Integer> integers = new Queue<Integer>();
    Queue<String> strings = new Queue<String>();
    Collection<Integer> intList = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20));
    Collection<String> strList = new ArrayList<>(Arrays.asList("xxx", "yyy",
"zzz"));
    boolean int_ret = integers.addAll(intList);
    boolean str_ret = strings.addAll(strList);
    System.out.print("integers.equals(strings) is " + integers.equals(strings)
+ "\n");
    Queue<Integer> others = (Queue<Integer>) integers.clone();
    System.out.print("integers.equals(others) is " + integers.equals(others) +
"\n");
    System.out.print("integers == others is " + (integers == others) + "\n");
    for (int i = 0; i < 20; i++) {
        Integer obj = integers.remove();
        System.out.print(obj + " ");
    }
    System.out.print("\n");

    System.out.print("integers.add(1) is " + integers.add(1) + "\n");
    System.out.print("integers.offer(2) is " + integers.offer(2) + "\n");
    try {
        System.out.print("integers.peek() is " + integers.peek() + "\n");
        System.out.print("integers.element() is " + integers.element() + "\n");
    } catch (Exception e) {
        System.out.print(e.toString() + "\n");
    }
    try {
        System.out.print("integers.poll() is " + integers.poll() + "\n");
        System.out.print("integers.remove() is " + integers.remove() + "\n");
    }
}

```

```

    } catch (Exception e) {
        System.out.print(e.toString() + "\n");
    }
    integers.addAll(intList);
    try {
        System.out.print("integers.offer(5) is " + integers.offer(5) + "\n");
    } catch (Exception e) {
        System.out.print(e.toString() + "\n");
    }
    try {
        System.out.print("integers.add(5) is " + integers.add(5) + "\n");
    } catch (Exception e) {
        System.out.print(e.toString() + "\n");
    }
    try {
        System.out.print("integers.offer(null) is " + integers.offer(null) +
"\n");
    } catch (Exception e) {
        System.out.print(e.toString() + "\n");
    }
    try {
        System.out.print("integers.add(null) is " + integers.add(null) + "\n");
    } catch (Exception e) {
        System.out.print(e.toString() + "\n");
    }
}
}
}

```