CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 7: Java Generics & Iterators

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin
JAVA GENERICS
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 1. Implement a separate stack class for each type.
   - Rewriting code is tedious and error-prone.
   - Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5.

(Java 1.5 released Sep 2004)
Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 1. Implement a separate stack class for each type.
   • Rewriting code is tedious and error-prone.
   • Maintaining cut-and-pasted code is tedious and error-prone.

@#$%! most reasonable approach until Java 1.5.
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 2. Implement a stack with items of type Object.
- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```java
StackOfObjects s = new StackOfObjects();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```
Parameterized stack

**We implemented:** StackOfStrings.

**We also want:** StackOfURLs, StackOfInts, StackOfVans, ....

**Attempt 3.** Java generics.
- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

```java
Stack<Apple> s = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

**Guiding principles.** Welcome compile-time errors; avoid run-time errors.
Generic stack: linked-list implementation

```java
public class LinkedStackOfStrings {
    private Node first = null;

    private class Node {
        String item;
        Node next;
    }

    public boolean isEmpty() {
        return first == null;
    }

    public void push(String item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop() {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

```java
public class Stack<Item> {
    private Node first = null;

    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty() {
        return first == null;
    }

    public void push(Item item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop() {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```
Generic stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        s = new String[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(String item) {
        s[N++] = item;
    }

    public String pop() {
        return s[--N];
    }
}
```

The way it should be:

```java
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity) {
        s = new Item[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(Item item) {
        s[N++] = item;
    }

    public Item pop() {
        return s[--N];
    }
}
```

@#$%^! generic array creation not allowed in Java
Generic stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public .StackOfStrings(int capacity) {
        s = new String[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(String item) {
        s[N++] = item;
    }

    public String pop() {
        return s[--N];
    }
}
```

```java
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity) {
        s = (Item[]) new Object[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public void push(Item item) {
        s[N++] = item;
    }

    public Item pop() {
        return s[--N];
    }
}
```

the way it is

the ugly cast
Unchecked cast

% javac FixedCapacityStack.java
Note: FixedCapacityStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
found    : java.lang.Object[]
required: Item[]
       a = (Item[]) new Object[capacity];
^          
1 warning

Q. Why does Java make me cast (or use reflection)?
Short answer. Backward compatibility.
Long answer. Need to learn about type erasure and covariant arrays.
Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.
- Each primitive type has a wrapper object type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast between a primitive type and its wrapper.

```java
Stack<Integer> s = new Stack<Integer>();
s.push(17); // s.push(Integer.valueOf(17));
int a = s.pop(); // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.
Generic Iterators
1.3 BAGS, QUEUES, AND STACKS

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
**Design challenge.** Support iteration over stack items by client, without revealing the internal representation of the stack.

<table>
<thead>
<tr>
<th>s[]</th>
<th>i</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>it</td>
<td>was</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Java solution. Make stack implement the `java.lang.Iterable` interface.
Q. What is an **Iterable**?
A. Has a method that returns an **Iterator**.

Q. What is an **Iterator**?
A. Has methods **hasNext()** and **next()**.

Q. Why make data structures **Iterable**?
A. Java supports elegant client code.

```
“foreach” statement (shorthand)
for (String s : stack)  
    StdOut.println(s);

equivalent code (longhand)
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```

```
java.lang.Iterable interface
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}

java.util.Iterator interface
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();  // optional; use at your own risk
}
```
Stack iterator: linked-list implementation

```java
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item> {
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item> {
        private Node current = first;

        public boolean hasNext() { return current != null; }
        public void remove() { /* not supported */ }
        public Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item> {
    ...

    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove() { /* not supported */ }
        public Item next() { return s[--i]; }
    }
}

<table>
<thead>
<tr>
<th>s[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

i

N
Iteration: concurrent modification

Q. What if client modifies the data structure while iterating?
A. A fail-fast iterator throws a java.util.ConcurrentModificationException.

```java
class ConcurrentModification
{
    for (String s : stack)
        stack.push(s);
}
```

Q. How to detect?
A.
- Count total number of `push()` and `pop()` operations in `Stack`.
- Save counts in `*Iterator` subclass upon creation.
- If, when calling `next()` and `hasNext()`, the current counts do not equal the saved counts, throw exception.
Generic Comparisons
Design Challenge: Add a search method in the Stack ADT.

```java
public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

    boolean search(Item searchObj)
    {
        // we need to compare searchObj to other items in the Stack
        ...
    }
}
```

Item needs to at least implement the comparable interface.
public interface Comparable<T> {
    int compareTo(T o);
    // Compares this object with objects of class T.
}

i.compareTo(o) returns:

→ 0 if i = o
→ >0 if i > o
→ <0 if i < o

Comparable is a parametric interface because it doesn’t know a priori the type T.
public class Stack<Item extends Comparable<Item>> {
    private Node first = null;

    private class Node {
        Item item;
        Node next;
    }

    ...  
    boolean search(Item searchObj) {
        for (Item i : this) {
            if (i.compareTo(searchObj) == 0) return true;
        }
        return false;
    }
}