04 A: Lists, Stacks, and Queues III

CS1102S: Data Structures and Algorithms

Martin Henz

February 3, 2010

Generated on Monday 1st February, 2010, 16:30

A Simple Box Class

```
public class IntegerBox {
    private Integer integer;
    public void add(Integer i) {
        integer = i;
    }
    public Integer get() {
        return integer;
    }
}
```

Is this practical?

Situation

Each time we want to have a box for some data type, we need to define a MyTypeBox class.

Is this practical?

Situation

Each time we want to have a box for some data type, we need to define a MyTypeBox class.

First Idea

Use Object as the type of the elements. After all, any Java object *is an* Object!

An Object Box Class

```
public class ObjectBox {
    private Object object;
    public void add(Object obj) {
        object = obj;
    }
    public Object get() {
        return object;
    }
}
```

Using the Object Box

```
public class ObjectBoxTest {
    public static void main(String[] args) {
        // ONLY place Integer objects into box!
        ObjectBox integerBox = new ObjectBox();
        integerBox.add(new Integer(10));
        Integer someInteger
        = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

Using the Object Box (2)

```
// ONLY place Integer objects into this box!
ObjectBox integerBox = new ObjectBox();

// Imagine this is one part of large application
// modified by one programmer.
integerBox.add("10"); // note type now String

// ... and this is another, perhaps written
// by a different programmer
Integer someInteger = (Integer)integerBox.get();
System.out.println(someInteger);
```

Is this practical?

Situation

In order to take items out of the box, we need to use a *cast* operation. This operation *circumvents* Java's type system, and is not safe, especially in large programs!

Is this practical?

Situation

In order to take items out of the box, we need to use a *cast* operation. This operation *circumvents* Java's type system, and is not safe, especially in large programs!

Idea

Write a "generic" class that can be re-used for any kind of content type

A Generic Box

```
public class Box<T> {
    private T t; // T stands for "Type"
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

Using A Generic Box

```
Box<Integer> integerBox = new Box<Integer>();
integerBox.add(new Integer(10));
Integer someInteger = integerBox.get(); // no cast!
System.out.println(someInteger);

// try adding a String:
// integerBox.add("some string");
```

How to write functions on boxes?

Issue

The Java operator new needs a proper class, not a generic one!

Example

```
public class BoxUtil {
    public static void
    fillBoxes(U u,
              List<Box<U>> boxes) {
        for (Box<U> box : boxes)
            box.add(u);
} }
```

How to write functions on boxes?

Example

```
public class BoxUtil {
    public static void
    fillBoxes(U u,
              List<Box<U>> boxes) {
        for (Box<U> box : boxes)
            box.add(u);
} }
```

Problem

Where does the type U come from?

Generic Method

Example

```
public class BoxUtil {
    public static <U> void
    fillBoxes(U u,
              List<Box<U>> boxes) {
        for (Box<U> box : boxes)
            box.add(u);
} }
```

Generic Method: Example

```
class Crayon {};
Crayon red = new Crayon();
List<Box<Crayon>> crayonBoxes
= new ArrayList<Box<Crayon>>();
...
BoxUtil.<Crayon>fillBoxes(red, crayonBoxes);
BoxUtil.fillBoxes(red, crayonBoxes);
// compiler infers that U is Crayon
```

An Issue: Subtyping

```
public class Subtyping {
    public static void someMethod(Number n){
        System.out.println(n);
    }
    public static void main(String[] args) {
        Object someObject = new Object();
        Integer someInteger = new Integer(10);
        someObject = someInteger; // OK
        someMethod(new Integer(10)); // OK
        someMethod(new Double(10.1)); // OK
} }
```

An Issue: Subtyping

```
public class Subtyping {
    public static void boxTest(Box<Number> n){
        System.out.println(n);
    }
    public static void main(String[] args) {
        Box<Number> box = new Box<Number>();
        box.add(new Integer(10)); // OK
        box.add(new Double(10.1)); // OK

        boxTest(new Box<Integer>()); // NOT OK
} }
```

Subtyping Explanation: Cages and Animals

```
public class Animal {}
public class Lion extends Animal {}
public class Butterfly extends Animal {}
public class Cage<E> extends HashSet<E>
                     implements Collection<E> {}
```

Subtyping Explanation: Cages and Animals

```
Lion king = new Lion();
Animal a = king;
Cage<Lion> lionCage = new Cage<Lion>();
lionCage.add(king);

Butterfly monarch = new Butterfly();
Cage<Butterfly> butterflyCage
= new Cage<Butterfly>();
butterflyCage.add(monarch);
```

Subtyping Explanation: Cages and Animals

```
Cage<Animal> animalCage = new Cage<Animal>();

animalCage.add(king);
animalCage.add(monarch);

animalCage = lionCage;  // compile-time error
animalCage = butterflyCage; // compile-time error
```

Squaring the Elements of a List

```
List<Integer> myIntegerList
= new ArrayList<Integer>();
myIntegerList.add(4);
myIntegerList.add(13);
myIntegerList.add(5);
List<Integer> myIntegerListSquare
= ListMap.integerListSquare(myIntegerList);
```

Squaring the Elements of a List

```
public class ListMap {
    public static List<Integer>
    integerListSquare(List<Integer> a) {
        ArrayList<Integer> resultList
        = new ArrayList<Integer>();
        for (Integer i : a)
            resultList.add(i * i);
            return resultList;
        }
}
```

Is this practical?

Situation

Each time we want to apply an operation on all elements of a list, we need to write a new loop.

Is this practical?

Situation

Each time we want to apply an operation on all elements of a list, we need to write a new loop.

Idea

We encode the operation to be applied on each element as an object.

Squaring the Elements of a List (2)

```
List<Integer> mySecondIntegerListSquare
= ListMap.integerListMap(myIntegerList,
                         new Square());
```

Squaring the Elements of a List (2)

```
public class ListMap {
    public static List<Integer>
    integerListMap(List<Integer> list,
                   IntegerToInteger f) {
        ArrayList<Integer> resultList
        = new ArrayList<Integer>();
        for (Integer i : list)
            resultList.add(f.apply(i));
        return resultList;
} }
```

Squaring the Elements of a List (2)

```
abstract class IntegerToInteger {
        abstract Integer apply(Integer i);
}
```

Squaring the Elements of a List (2)

```
public class Square extends IntegerToInteger {
        public Integer apply(Integer x) {
                return x * x;
        }
}
```

Is this practical?

Situation

Each time we want to apply an operation on all elements of a list, we need to write a new class for the operation (e.g. Square).

Is this practical?

Situation

Each time we want to apply an operation on all elements of a list, we need to write a new class for the operation (e.g. Square).

Idea

Place the class where it is needed!

Squaring the Elements of a List (3)

```
List<Integer> myThirdIntegerListSquare
= ListMap.integerListMap(
      myIntegerList,
      new IntegerToInteger(){
            Integer apply(Integer i) {
               return i * i;
            }
         });
printIntList(myThirdIntegerListSquare);
```

Is this practical?

Situation

Each time we want to apply an operation on all elements of a list, we need to write a new class for the function type (e.g. IntegerToInteger)

Is this practical?

Situation

Each time we want to apply an operation on all elements of a list, we need to write a new class for the function type (e.g. IntegerToInteger)

Idea

Make the function generic!

Squaring the Elements of a List (4)

```
abstract class IntegerToInteger {
        abstract Integer apply(Integer i);
}
abstract class Function<Any> {
        abstract Any apply(Any i);
}
```

Squaring the Elements of a List (4)

```
List<Integer> myFourthIntegerListSquare
= ListMap.listMap(
      myIntegerList,
      new Function<Integer>(){
            Integer apply(Integer i) {
               return i * i;
            }
      });
```

Another Example: Function Composition

```
(define (compose f g)
   (lambda (x) (f (g x))))
(define square-after-inc (compose square inc))
(square-after-inc 6)
```

Another Example: Function Composition

```
public static <Any> Function<Any>
    compose(final Function<Any> f,
            final Function<Any> g) {
        return new Function<Any>(){
                    Any apply(Any i) {
                        return f.apply(g.apply(i));
                    }
                };
        }
```

Another Example: Function Composition

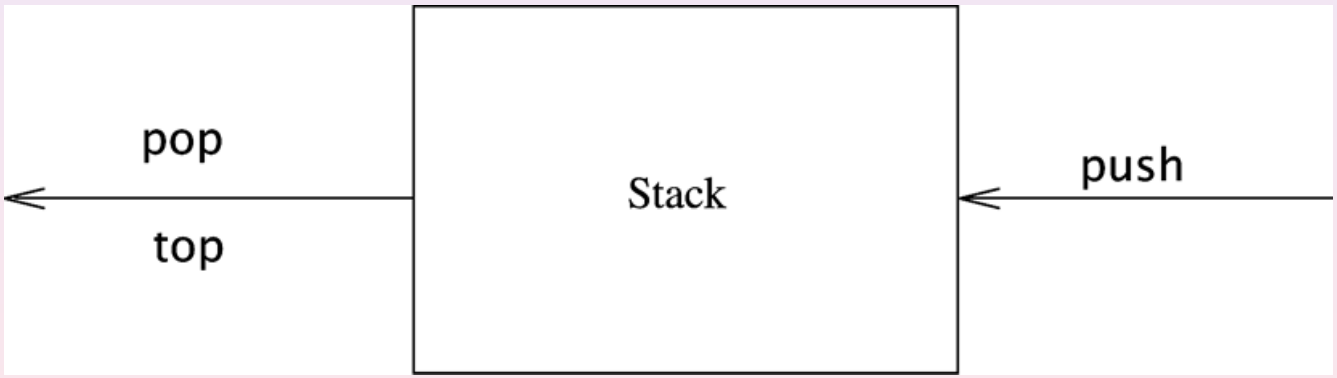
```
Function<Integer> squareAfterInc
= Function.compose(
      new Function<Integer>(){
            Integer apply(Integer i) {
               return i * i; } },
      new Function<Integer>(){
            Integer apply(Integer i) {
               return i + 1; } });

System.out.println(squareAfterInc.apply(6));
```

Motivation

Purpose of stacks

Collections that serve as intermediate storage of data items



Stack Model

Stack access

Only the top element of a stack is accessible through top and pop operations

Stack Model

Stack access

Only the top element of a stack is accessible through top and pop operations

Stack discipline

Last in—first out: LIFO

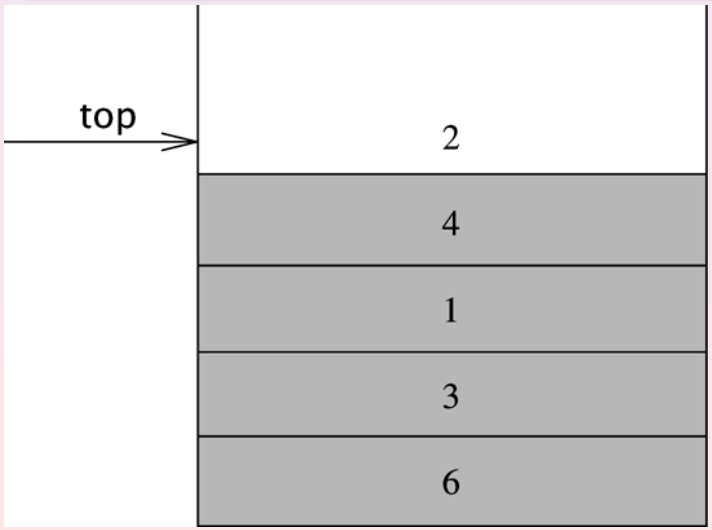
Stack Model

Stack access

Only the top element of a stack is accessible through top and pop operations

Stack discipline

Last in—first out: LIFO



Implementation of Stacks using ArrayList

```
class ArrayStack<E> extends ArrayList<E> {
  public boolean empty() { return size() == 0; }
  public E push(E item) { add(item); return item;}
  public E top() {
    if (empty())
      throw new java.util.EmptyStackException();
    else return get(size()-1); }
  public E pop() {
    if (empty())
      throw new java.util.EmptyStackException();
    else return remove(size()-1); } }
```

Example of Usage

```
Stack<Task> s = new ArrayStack<Task>();
s.push(new Task(...));
s.push(computeSomeTask(...));
s.top().displayTask();
s.pop().completeTask();
```

Implementation of Stacks using LinkedList

```
class LinkedListStack<E> extends LinkedList<E> {
  public boolean empty() { return size() == 0; }
  public E push(E item) {
    add(item,0); return item; }
  public E top() {
    if (empty())
      throw new java.util.EmptyStackException();
    else return get(0); }
  public E pop() {
    if (empty())
      throw new java.util.EmptyStackException();
    else return remove(0); } }
```

Stacks in Practice

- Stack classes are so similar to List classes that often Lists are used directly as stacks.
- Example:

```
Stack<Task> s = new LinkedList<Task>();
s.add(0,new Task(...));
s.add(0,computeSomeTask(...));
s.get(0).displayTask();
s.remove(0).completeTask();
```

Balancing symbols

Check whether the parentheses in expression are properly nested; parentheses occur in pairs around proper expressions.

Algorithm

```
Stack<Character> s = new Stack<Character>();
while (!empty(input)) {
    Character c = input.nextCharacter();
    if (closingPar(c)) {
        if (s.empty()) error("not␣proper");
        else checkPar(s.pop(),c);
    } else if (openingPar(c))
        s.push(c);
}
```

Expression Evaluation

Infix notation

Easy for humans, but need precedences, parentheses etc.

Example

$6 * (5 + (2 + 3) * 8 + 3)$

Postfix notation

Place operator after arguments; no ambiguities or parentheses

Example

$6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *$

Evaluation of Postfix Expressions

Idea

Keep intermediate values in a stack.

Algorithm

Proceed from left to right

- number: push on stack
- operator: pop first two numbers from stack, apply operation on them, and push result back onto stack

Operand Stack in JVM

```
2 + 3 * 8
```

is compiled to

```
bipush 2
bipush 3
bipush 8
imul
iadd
```

Method Calls

Runtime stack

Data structure to keep track of the state of the JVM that needs to be restored when methods return

Runtime stack frames

- Program counter (code address to be returned to)
- Local variable area (for arguments and local variables)
- Operand stack (for intermediate values)

Example

```
int i = 5;
System.out.println(f(7) + i);
```