CSC207 - Java Interfaces

Ilir Dema

Summer 2016

Interfaces

- In computing, an interface is shared boundary across which two separate components of a computer system exchange information.
  - The exchange of information can be described in terms of behaviors (i.e. methods).
  - None of the systems involved "cares" how this information is produced.
  - Which means the description of the behaviours has to be abstract.

Java Interfaces

- A java Interface is similar to a Java class
    - can include variable declarations
    - can include methods
- However
    - Variables must be constants
    - Methods must be abstract.
- A Java interface cannot be instantiated.
- We can use an interface to formally specify the logical level of an ADT:
- It provides a template for classes to fill.
- A separate class then ”implements” it.

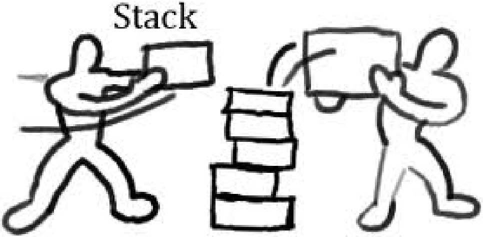
Advantages

- We can formally check the syntax of our specification. When we compile the interface, the compiler uncovers any syntactical errors in the method interface definitions.
- We can formally verify that the interface contract is met by the implementation.
- When we compile the implementation, the compiler ensures that the method names, parameters, and return types match what was defined in the interface.
- We can provide a consistent interface to applications from among alternate implementations of the ADT.

Inheritance of interfaces

- A Java interface can extend another Java interface, inheriting its requirements.
- If interface B extends interface A, then classes that implement interface B must also implement interface A.
- Usually, interface B adds abstract methods to those required by interface A.

Example: Stack ADT



Stack: A structure in which elements are added and removed from only one end; a last in, first out (LIFO) structure

Generic collections

- Collection is an object that holds other objects.
  - Typically we want to perform the following operations on a collection:
    - inserting
    - removing
    - iterating through the contents in a non-destructive fashion.
- A stack is an example of a collection ADT.
  - It collects together elements for future use, while maintaining a "last in, first out" ordering among the elements.
- The particular operations for stacks are:
  - push: inserts an element
  - pop: removes an element
  - top: allows access to the element sitting on top of the stack
- None of these should depend on the data type of the elements stored in the stack!
- That means stack is a generic collection.

Defining the behaviour of stack abstract methods

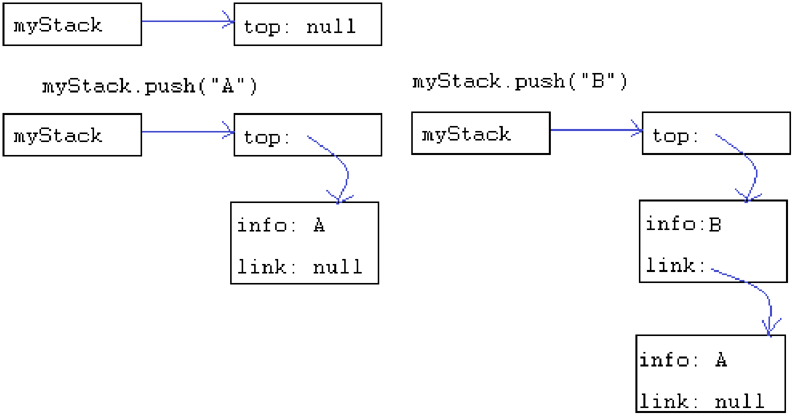
- **push**: insert an element on the top of the stack. The new element becomes top of the stack.
  - We can push an element if the container holding the stack elements is capable of holding another element.
    - What happens if the container is full? We need to throw an **exception**!
    - Alternatively, we can have an undbounded container. No need to throw an exception.
- **pop**: remove an element from top of the stack. If the stack is empty, we need to throw an exception.
- **top**: return a reference to the object sitting on top of the stack. If the stack is empty, we need to throw an exception.

Defining the interfaces

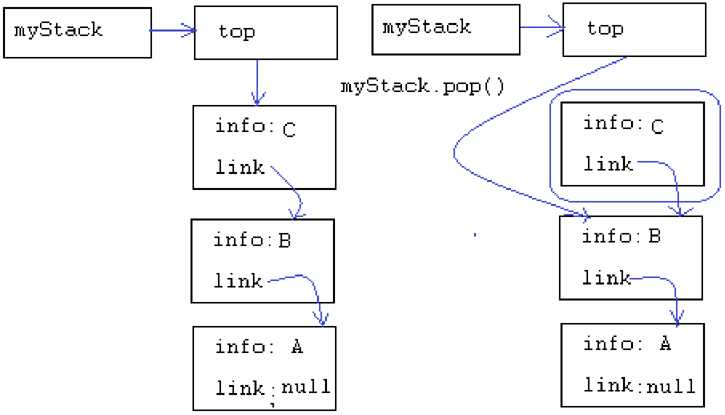
- **pop** and **top** behave same way regardless of the nature of the stack container.
- That means we can define a `StackInterface` containing `pop()` and `top()`, both throwing `StackUnderflow` exception
- Next, we can extend this interface in two ways:

  `UnboundedStackInterface` containing a `push` method
  `BoundedStackInterface` containing a `push` method that throws `StackOverflow` exception.
- Notice the concrete nature of the stack elements is unimportant at this point.
- However for the purpose of defining the methods, we need a placeholder for the data type of the stack elements.
- We will **parametrize** the interface by letting a dummy parameter denoted by `T` stand for any data type that we want to store in our stacks.

The effect of push



The effect of pop



Before pop

After pop

(Node C is available to garbage collector)

UnboundedStack implementation

- For an unbounded implementation, we need a source of nodes, capable to hold a reference to a generic data object and a link to the next node.
- The nodes will be instances of our class `Node`, member of `support` package.
- Our class that implements `UnboundedStack` interface will be called `LinkedStack` since we litellarly will maintain links between nodes that compose the stack elements.
- We also need to define checked exception class `StackunderflowException`.
- Our implementation needs only one class variable `top` that points to the top of the stack.
- We will also add a public boolean method `isEmpty` (an observer) that returns true if our stack is empty.

Application: well-formed expresssions

- Given a paired set of grouping symbols, determine if the open and close versions of each symbol are matched correctly.
- Examples: (), [], $\langle\rangle$, etc.
- Any number of other characters may appear in the input expression, before, between, or after a grouping pair, and an expression may contain nested groupings.
- Each close symbol must match the last unmatched opening symbol and each open grouping symbol must have a matching close symbol.

Well-formed expressions algorithm

```
returns a string indicating the result
instantiate a stack myStack
for each ch in inputString:
    if (ch isOpenSymbol) myStack.push(ch)
    if (ch isClosingSymbol)
        if (myStack.isEmpty()) return ”unbalanced”;
        if (ch isNotEqual(ClosingSymbolOf(myStack.top())) return ”unbalanced”;
        myStack.pop()
if not(myStack.isEmpty()) return "premature end"
else return "well formed";
```