Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Upcoming Schedule

- **Lecture 7** (today)
  - Array-based Stack
- **Lecture 8** (next lecture)
  - Linked Stack, Postfix expression
- **Next week**
  - Tuesday Feb 16: (Mon schedule)
  - Wednesday Feb 17: first mid-term 7-9pm ILC N151

Today's topics

- The `Iterable` and `Iterator` Interfaces
- The Stack Interfaces
- Array Stack Implementation

Iterator

- An **`iterator`** is an object that allows you to traverse the elements in a collection one-by-one, regardless of how the collection is implemented.

- The **`iterator`** interface has three methods:

  - **`hasNext()`**: returns true if the collection has more elements to traverse, false otherwise.

  - **`next()`**: returns the next element. To get all elements, call this repeatedly. If there is no more element (`hasNext()` returns false), this method throws `NoSuchElementException`.

  - **`remove()`**: removes the last returned element.

Iterator and Iterable

- Example (let's say object **`list`** stores a collection of **type T** objects, doesn't matter how stored)

```
Iterator<T> iter = list.iterator();
while (iter.hasNext()) {
  T element = iter.next();
  … …
}
```

- To do so, `list` must be an instance of a class that implements the **`Iterable<T>`** interface, which contains just one method that returns the iterator:

```
Iterator<T> iterator();
```

Putting it Together

- Let's say `List<T>` is a generic class that stores a collection of type `T` objects, and it implements the `Iterable<T>` interface:

```
class List<T> implements Iterable<T> {
  public Iterator<T> iterator() {
    return new ListIterator<T>(…);
  }
  // variables, constructors, other methods
}
```

- Note the `iterator()` above returns a `ListIterator<T>` object, which implements the `Iterator<T>` interface. It might look like this:

Putting it Together

```
class ListIterator<T> implements Iterator<T>
{
  public boolean hasNext() {…}
  public T next() {…}
  public void remove() {…}
  // variables, constructors, other methods
}
```

- Here `ListIterator<T>` is aware of the specific implementation details of `List<T>` and provides the above methods to traverse the collection of objects.
- Reference to the `List<T>` object is typically passed in through the constructor.

Putting it Together

```
List<String> list = new List<String>();
… …
Iterator<String> ite = list.iterator();
// ite is of class ListIterator<String>
while (iter.hasNext()) {
  String e = iter.next();
  … …
}
```

- In fact, there is an easier way to traverse the list:

```
for(String e : list) {
  … …
}
```

Summary

```
List<String> list = new List<String>();
… …
for(String e : list) {
  … …
}
```

- A class (e.g. List) can implement the **`Iterable`** interface. If so it must implement the **`iterator()`** method that returns an iterator object.
- The iterator implements the **`Iterator`** interface, and provides the **`hasNext()`**, **`next()`**, and **`remove()`** methods.
- To traverse the elements in an iterable object (e.g. a List object), you can use the **`for( : )`** loop as above.

The Two Stack Interfaces

- DJW define two Stack interfaces in order to have both **bounded** and **unbounded** stacks.

- `BoundedSI<T>` and `UnboundedSI<T>` differ in that only `BoundedSI<T>` has an `isFull` method, and they have different `throws` clauses.

- Note that DJW's `pop()` does **not** return the element popped -- you have to get it with "top" first if you want to save it. This is done to completely separate transformers with observers.

The Stack Interfaces

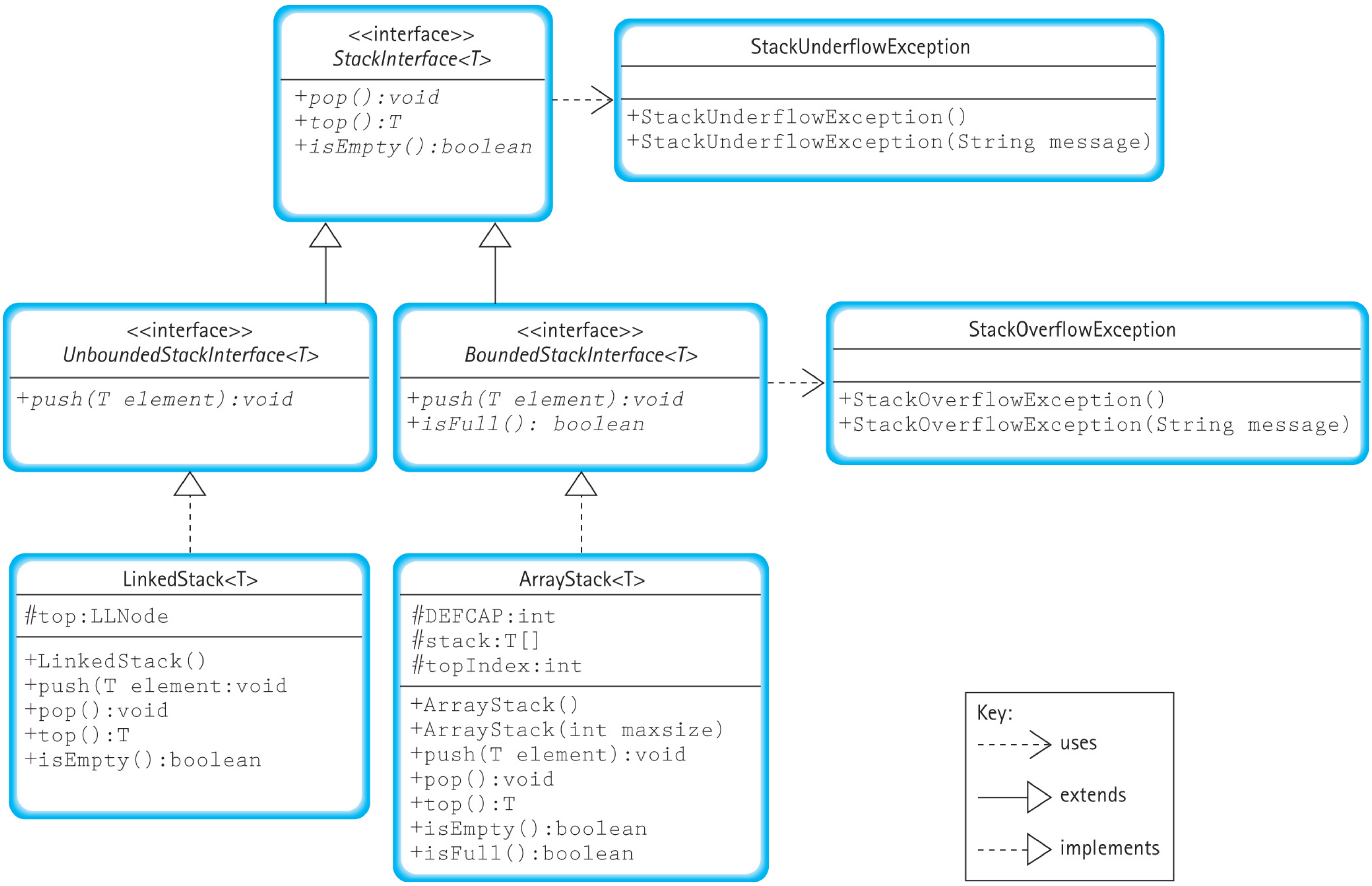
```
public interface StackInterface<T> {
    void pop( ) throws StackUnderflowException;
    T top( ) throws StackUnderflowException;
    boolean isEmpty( );
}
```

The Stack Interfaces

```
public interface StackInterface<T> {
    void pop( ) throws StackUnderflowException;
    T top( ) throws StackUnderflowException;
    boolean isEmpty( );
}

public interface BoundedSI<T> extends StackInterface<T> {
    void push(T element) throws StackOverflowException;
    boolean isFull( );
}

public interface UnboundedSI<T> extends StackInterface<T>
{
    void push(T element);
}
```


<<interface>>
StackInterface<T>
+pop():void
+top():T
+isEmpty():boolean
StackUnderflowException
+StackUnderflowException()
+StackUnderflowException(String message)
<<interface>>
UnboundedStackInterface<T>
+push(T element):void
<<interface>>
BoundedStackInterface<T>
+push(T element):void
+isFull(): boolean
StackOverflowException
+StackOverflowException()
+StackOverflowException(String message)
LinkedStack<T>
#top:LLNode
+LinkedStack()
+push(T element:void
+pop():void
+top():T
+isEmpty():boolean
ArrayStack<T>
#DEFCAP:int
#stack:T[]
#topIndex:int
+ArrayStack()
+ArrayStack(int maxsize)
+push(T element):void
+pop():void
+top():T
+isEmpty():boolean
+isFull():boolean
Key:
uses
extends
implements

Idea of the Implementation

- In our first Stack implementation, we will use an array as the underlying storage structure, and name it **`ArrayStack<T>`**.
- Since an array has fixed size, **`ArrayStack<T>`** will implement the **`BoundedStackInterface`** rather than **`UnboundedStackInterface`**.

Exceptions in ArrayStack

- **`push()`** will throw a **`StackOverflowException`** if it's called when the array is already full.
  - **`isFull()`** method is provided to allow the user to check before calling **`push()`**.
- **`pop()`**, **`top()`** will throw **`StackUnderflowException`** if they are called when the stack is empty.
  - **`isEmpty()`** method is provided to allow the user to check before calling **`pop()`** or **`top()`**.

Exceptions in ArrayStack

- In the textbook, both `StackOverflowException` and `StackUnderflowException` are classes that extends Java's **`RuntimeException`** class.

- Such classes are called **unchecked** exceptions, meaning methods that throw such exceptions do not need to declare them in the signature, and callers are not required to `try-catch` them.

- Some common unchecked exceptions include `NullPointerException`, `IndexOutOfBoundException`, `NoSuchElementException`.

Exceptions in ArrayStack

- Other exceptions (such as extended from the `Exception` class) are called **checked** exceptions.

- Methods that throw such exceptions must declare them in the signature; and they must be explicitly handled, meaning callers must either wrap these methods in `try-catch` statement, or throw it further up.

- A common checked exception is `IOException`.

- As confusing as it may sound, `RuntimeException` itself is a subclass of `Exception`.

Code for the Data Fields

```
public class ArrayStack<T> implements
    BoundedStackInterface<T> {
    // default capacity
    protected final int DEFCAP = 100;
    // holds stack elements
    protected T[ ] stack;
    // index of top element
    protected int topIndex = -1;
}
```

- We have just three data fields, a constant, an array, and index (same as in `ArrayStringLog`).

Data Fields for ArrayStack

- As before, the convention we use here is: **`topIndex`** points to the element at the top of the stack. It's initialized to **`-1`** indicating the stack is empty at the moment.

- When **`topIndex`** is equal to **`stack.length-1`**, it has reached the capacity of the array, hence the stack is full.

Observers for ArrayStack

```
public boolean isEmpty( ) {
    return (topIndex == -1);
}
public boolean isFull( ) {
    return (topIndex == (stack.length-1));
}
public int size( ) {
    return topIndex + 1;
}
```

Constructors for ArrayStack

- **Conceptually**, we want to do this:

```
public class ArrayStack<T> implements
     BoundedStackInterface<T> {
     … …
     public ArrayStack( ) {
        stack = new T [DEFCAP];
     }
}
```

Constructors for ArrayStack

- **Conceptually**, we want to do this:

```
public class ArrayStack<T> implements
    BoundedStackInterface<T> {
    …  …
    public ArrayStack( ) {
      stack = new T [DEFCAP];
    }
}
```

- **However**, Java **does not** allow you to create a generic array (i.e. with a to-be-decided type).

Casting to a Generic Type

- The work-around is to create an array of type **`Object`** (i.e. the super-class of all Java objects), and **explicitly cast** the resulting array into type **`T[]`**.
    - The compiler will warn us about Type Safety (that it cannot be sure the elements stored in the array are of type T), but we will ignore this warning.
- This makes sense because in an array of objects, each element is merely a reference (pointer). Whether it's an array of Strings, Apples, Objects, the array takes the same amount of memory space.

Code for the Constructors

```
public ArrayStack( ) {
    stack = (T[ ]) new Object[DEFCAP];
}

public ArrayStack(int maxSize) {
    stack = (T[ ]) new Object[maxSize];
}
```

- As before we have two constructors, one with no parameter that uses the default capacity, and one with a custom capacity as parameter.

Code for **push**

```
public void push (T element) {
    if (!isFull( )) {
        topIndex++;
        stack[topIndex] = element;
    } else throw new StackOverflowException
        ("push to full stack");
}
```

The `pop()` method

- Remember that DJW, unlike other stack implementations, separate the transformer `pop` from the observer `top`.
    - `pop` just discards the top element,
    - `top` returns the top element
- We'll implement the methods using DJW's vocabulary.

Code for `pop()`

```
public void pop( ) {
   if (!isEmpty( )) {
      stack[topIndex] = null; // releases reference
      topIndex--;
   } else throw new StackUnderflowException
      ("pop empty stack");
}
```

Code for `top()`

```
public T top( ) {
    T topOfStack = null;
    if (!isEmpty( ))
        topOfStack = stack[topIndex];
    else throw new StackUnderflowException
        ("top of empty stack");
    return topOfStack;
}
```

Java’s `ArrayList` Class

- It would be nice if the array capacity is not fixed!
- At the end of the Linked List lecture, we briefly mentioned arrays that can dynamically expand, thus overcoming the ‘fixed capacity’ limitation.
- Java provides several variable-length generic array structures, such as **`ArrayList<T>`**. It begins with some fixed capacity, but the capacity is reached, it copies itself into another array of twice the size, thus *appears* to be an array of unlimited size.

Java’s `ArrayList` Class

- What’s this doubling-the-capacity business?
  - Let’s say we have an initial ArrayList of capacity 10, and we keep adding elements to it.
  - Adding the 11th element causes the ArrayList to allocate a new array of capacity 20, copy the existing 10 elements and the 11th element to the new array, then release the old array.

Java's `ArrayList` Class

- What's this doubling-the-capacity business?
  - Let's say we have an initial ArrayList of capacity 10, and we keep adding elements to it.
  - Adding the 11th element causes the ArrayList to allocate a new array of capacity 20, copy the existing 10 elements and the 11th element to the new array, then release the old array.
  - Same thing happens as the 21th element is added.
  - By the time we have 81 elements we have done four capacity 'expansions': to 20, 40, 80, and 160.

The `ArrayListStack` Class

- You can use Java's **`ArrayList`** to implement Stack. Then it won't be bounded and hence can implement the **`UnboundedStackInterface`**.

- DJW give code for an **`ArrayListStack<T>`** class on pages 192-3. It uses the **`add`** and **`remove`** methods of **`ArrayList`** to implement **`push`** and **`pop`** respectively, and its code is otherwise similar to that of **`ArrayStack<T>`**.