CS206

Assignment 3 recap,

Stacks,

Debugging

A3

- "Not tight" on try-catch

```
Scanner fScanner=null;
try {
    fScanner = new Scanner(new File("/A/file/name"));
}
catch (FileNotFoundException fnfe)
{
    System.err.println("The file does not exist " + "/A/file/name");
    System.exit(0);
}
// do stuff
if (fScanner!=null)
    fScanner.close();
```

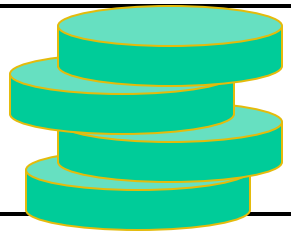
- Integer.parseInt & Double.parseDouble
  - can throw exception, no one caught
- src/README
  - some had README in the wrong place

A3

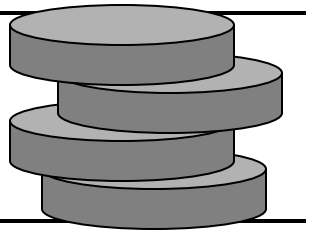
- super.toString()
  - could have / should have used
- Static methods — never again

```
public class Places
{
    private ArrayList<Place> theList;
    public Places() {
     theList = new ArrayList<>();
    }
    private Place parseLine(String line) {
     // do then return something
    }
    public void readZipCodes(String fileName) {
     // do stuff
    }
    public Place lookupZip(String zipCode) {
     // do then return something
    }
}
```

This is one alternative to static methods



Stacks



- Stack ADT stores abstract objects
- Insertion and deletions are First In Last Out – FILO
- Spring-loaded plate dispenser

- Operations
  - `push(Object)`
  - `Object pop()`
  - `Object top()`
  - `int size()`
  - `boolean isEmpty()`

`Stack` Interface

- Interface describing the stack ADT
- `null` is returned from `top()` and `pop()` when stack is empty
- Different from `java.util.Stack`
  - peek not top
  - no size()

```
public interface
Stack<E> {
  int size();
  boolean isEmpty();
  E pop();
  E top();
  void push(E element);
}
```

Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Exception vs Returning `null`

- We allow operations `pop` and `top` to be performed even when the stack is empty, by returning `null`
- Option: throw an exception
- Java uses both techniques
  - generally, throwing exception is more agressive & uses more system resources. So used only when there is really an error.

Array-based Stack

- Implement the stack with an array
- Add elements onto the end of the array
- Use an int `size` to keep track of the top

```
int size() {return size+1;}
E pop(){
  if (isEmpty()){return null;}
  else {size=size-1; return S[size+1];}
}
```

Push

- Array has set size and may become full
- A `push` will throw an exception if the array becomes full
  - Limitation of the array-based implementation
  - Linked list does not suffer from this
  - Alternatives? Growth a la ArrayList? Why not?

```
void push(E e) {
  if (size==S.length-1) {
    throw new
IllegalStateException();
  }
  else {
    size = size+1;
    S[size] = e;
  }
}
```

Performance and Limitations

- Performance
  - let n be the number of objects in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - Max size is limited and can not be changed
  - Pushing onto a full stack results in an implementation-specific exception

Code

```
public class ArrayStack<E> implements StackInterface<E>{
    private static final int CAPACITY = 1000;
    private E[] theStack;
    private int theSize;
    @Override
    public int size() {
    return theSize;
    }
    @Override
    public boolean isEmpty() {
    return theSize==0;
    }
    @Override
    public E pop() {
    if (theSize==0) return null;
    return theStack[–theSize];
    }
    @Override
    public E top() {
    return theSize==0 ? null : theStack[theSize];
    }
```

Code

```
    @Override
    public void push(E element) throws IllegalStateException {
    if (theSize==CAPACITY)
        throw new IllegalStateException("Stack Full!");
    theStack[theSize++]=element;
    }
    public ArrayStack() {
    this(CAPACITY);
    }
    @SuppressWarnings("unchecked")
    public ArrayStack(int capacity) {
    theStack = (E[])new Object[capacity];
    }
}
```

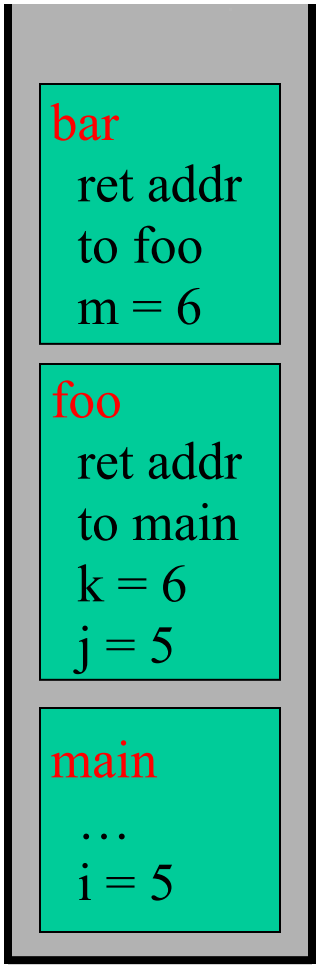
Method Stack in the JVM

- The JVM keeps track of the chain of active methods with a stack
  - printStackTrace()
- On a method call, the JVM pushes onto the stack a frame containing:
  - parameters
  - local variables
  - return address
- When a method ends, control passes onto the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  …
}
```



Stack Applications

- Reversing
- Palindromes
  - Madam Im adam
  - A man a plan a canal panama!

Stack Applications: Postfix

- Postfix notation
  - 5 6 * 2 + = (5*6) + 2
  - 3 4 5 * – = 3 – 4 * 5
  - 3 4 – 5 * = (3 – 4) * 5
- Evaluating postfix expressions with a stack
  - operands – push
  - operator – pop top two operands, perform operation and push results back on

Debugging in Eclipse

- Setting Breakpoints
- View Stack
- View Var values

http://www.vogella.com/tutorials/EclipseDebugging/article.html