CS151 Intro to Data Structures

Checkstyle

Stacks

JUnit

Announcements

HW1 Autograder issues

- HW02 due released
  - Due Thursday Feb 15th
  - START EARLY
- New Office Hours
  - Wednesday after class (Wednesday 2:30-3:30pm)
  - New policy:
    - if you can't make my office hours AND you have asked TAs for help AND you have spent >3-5 hours on the ***same bug***
    - you can email me a zip with your code. If you include a list of everything you have tried so far, I will debug

Announcements

HW1
Extension until Thursday

Things to check

1. Named your methods exactly as written in the assignment
2. Constructor types
3. don’t have a try-catch in main or LookupZip

Let’s take a minute now to fix

Outline

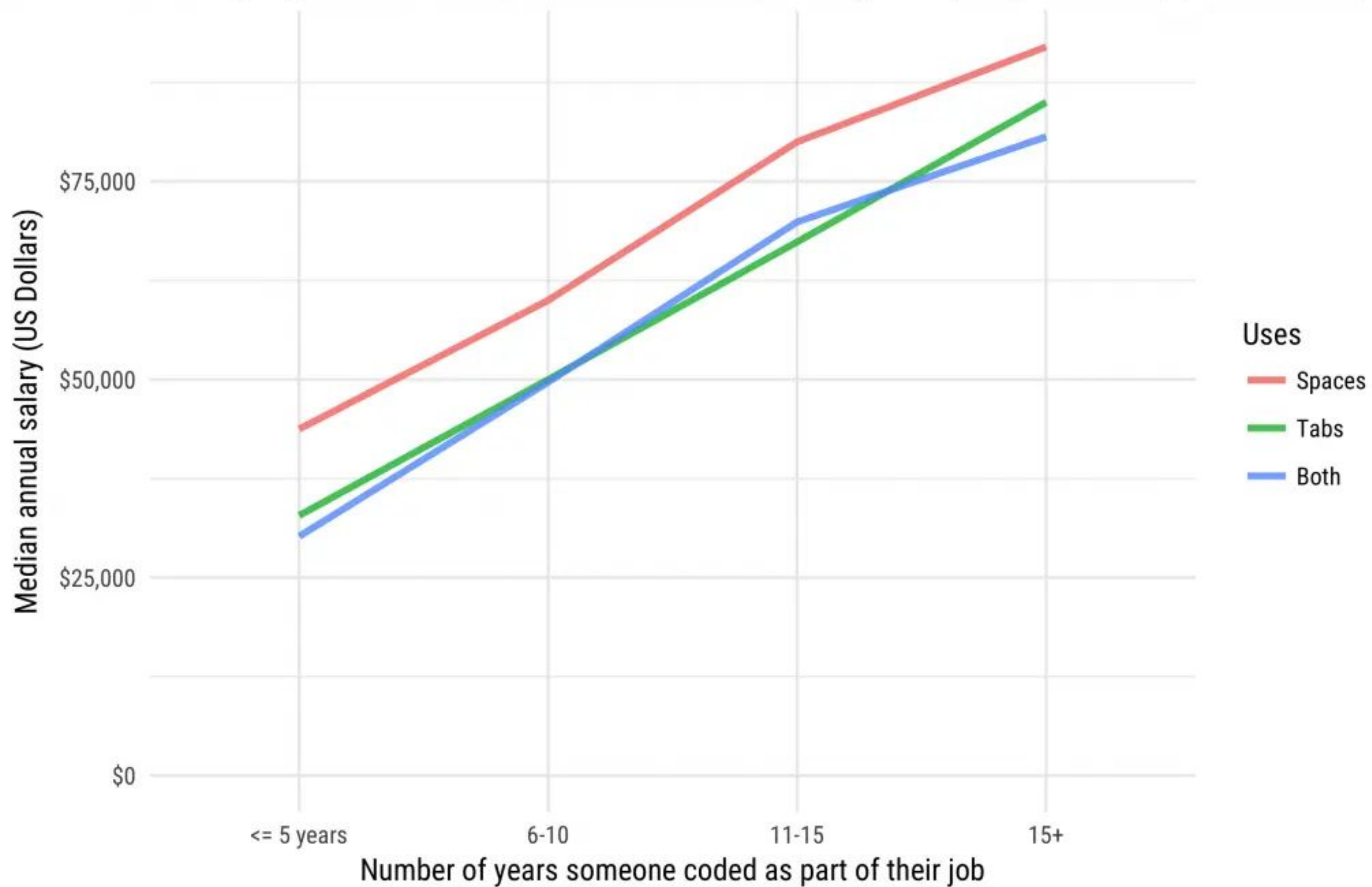
- Checkstyle
- Stacks
- JUnit

Checkstyle

- A tool that ensures that the code adheres to a specified coding standard
- Helps with readability
  - Especially coding in teams
- Checkstyle is **linter** static code analysis tool used to find and flag stylistic issues, and other issues in source code

CS151 Style Requirements

- Each file < 2000 lines
- No tabs
  - Tabs can be a different size on different computers and printers.
  - With spaces, the code will look the same regardless of the computer.
  - Our vim setup will automatically replace tabs with spaces
- Javadocs required
  - classes
  - methods
  - public and protected instance variables
  - must be properly formatted
- global const / final variable names must be all caps
- local variables and methods must be **camelcase**
- No star imports (import java.util.*)
- No unused imports
- Lines must be under 80 characters
- Methods must be under 150 lines
- and more :)


Salary differences between developers who use tabs and spaces
From 12,426 professional developers in the 2017 Developer Survey results, who provided tabs/spaces and salary
Median annual salary (US Dollars)
$75,000
$50,000
$25,000
$0
<= 5 years
6-10
11-15
15+
Number of years someone coded as part of their job
Uses
Spaces
Tabs
Both

Javadoc review

Proper format for a class

- /** .... **/
- First sentence must end in a period

```
/**
*  Description of what this segment of code does.
**/
```

Javadoc review

Proper format for a method

```
/** A method to return 0.
* @return an integer 0
* @param foo an input
**/
public int retZero(int foo) {
    return 0;
}
```

First sentence ends with a period

If there is a return value, must add a `@return` flag.

If there are any parameters, they must be marked with a `@param` flag followed by the param name

Checkstyle Example

```
java -jar checkstyle-8.16-all.jar -c cs151_checks.xml LookupZip.java
```

Checkstyle Requirements

HW03 onwards, all code must comply with the checkstyle

Future courses require it

Employers often require it

Makes working collaboratively much easier

Stacks

Stacks - FILO

- First In Last Out

- *stack* of plates in the dining hall

- Operations:
  - push
  - pop
  - peek
  - isEmpty

Stack Interface

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  E pop();
  E peek(); //does not modify the stack
  void push(E element); //pushes to top of stack
}
```

Stack Example - Browser History

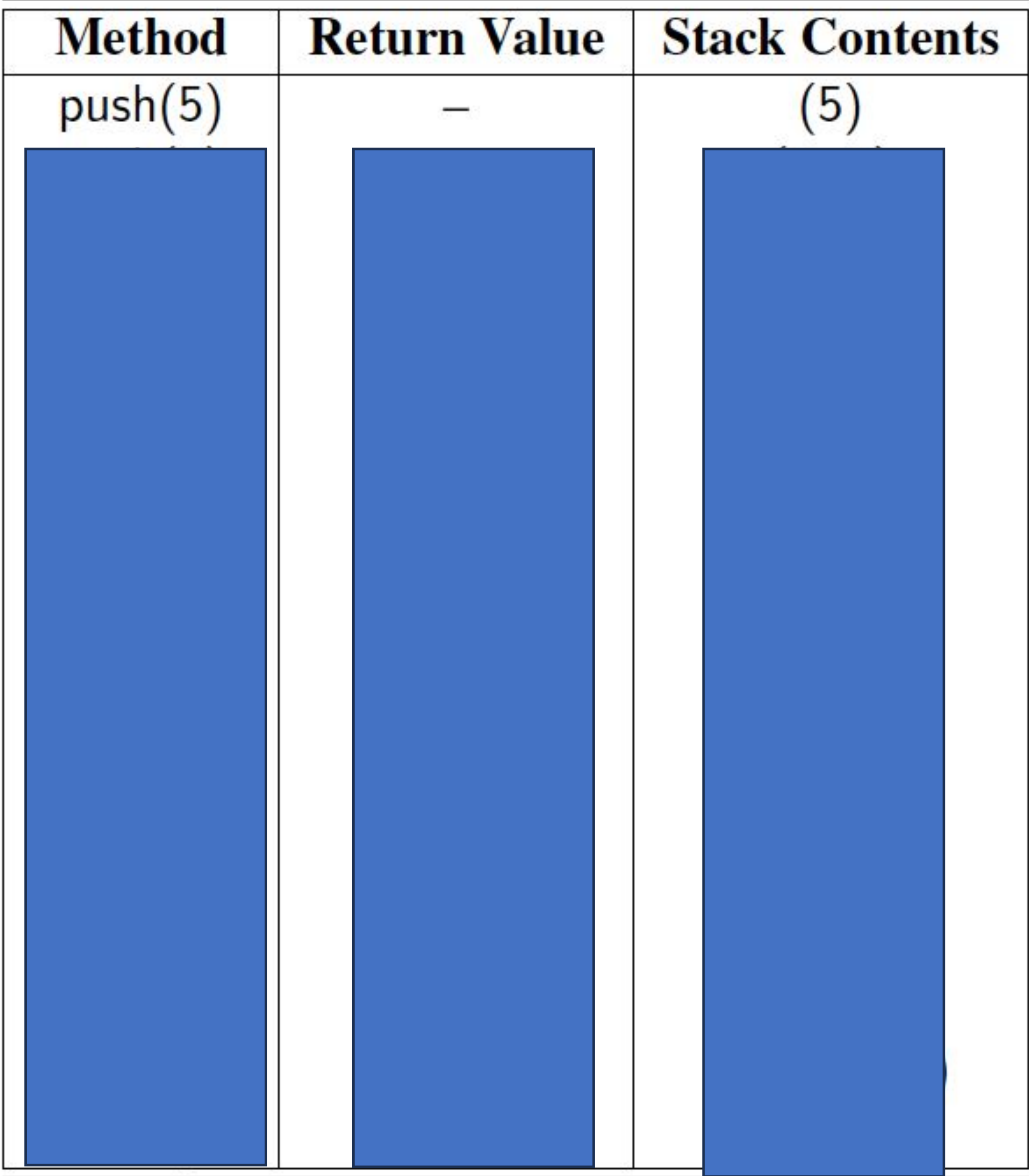
- Coding using the Java Stack data structure (we won't implement our own yet)

- What happens if you call pop on an empty stack?
  - This is called a precondition.

Stack Example

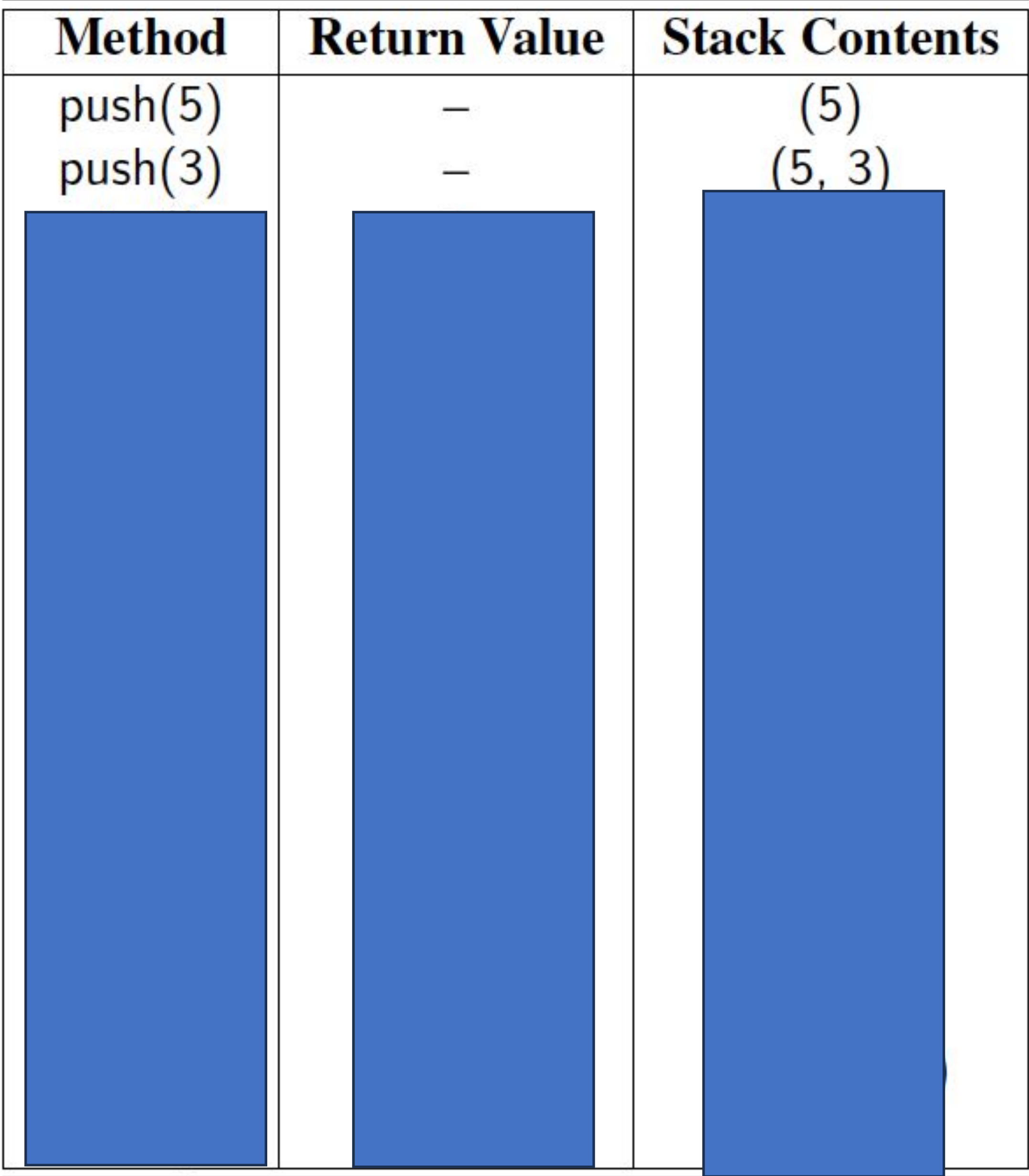
Method	Return Value	Stack Contents
push(5)		

Stack Example



Method	Return Value	Stack Contents
push(5)	–	(5)

Stack Example



Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)

Stack Example

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

Stack 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)

Another Stack Example - Calculator

Postfix notation: A way to represent mathematical operations where *operators* come after their *operands*

- 3 4 +

Infix notation (normal): *operators* between *operands*

- 3 + 4

Postfix notation eliminates the need for parenthesis

Postfix Notation - Examples

What is the postfix notation of (3 * 7)?

- 3 7 *
- How would we use a stack for this? Let’s draw it out
- Now, let’s code it

Postfix Notation - Examples

What is the postfix notation of (3 * 4 * 5)

- 3 4 5 * *
- Draw this out on board

What is the postfix notation of (3 + 4 * 5). Think PEMDAS! Which operation should occur first. We read left to right

- 3 4 5 * +
- Draw this out on board!

Call Stacks

- Manages the flow of control in a program

- Data structure (stack) that stores the information about the current method

- As functions call and return, the call stack grows and shrinks

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:        int k = (int) x;
506:        return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:        int k = (int) x;
506:        return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:        int k = (int) x;
506:        return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:      int y = 42;
102:      int z = 13;
          …
400:      int x = g(13 + a);
          …
497:      return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:      int val = f(15);
510:      System.out.println(val);
511:   }
```

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:        int k = (int) x;
506:        return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

h
Return val to g line 501
x=56
k=56
val = 57

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:      int y = 42;
102:      int z = 13;
          …
400:      int x = g(13 + a);
          …
497:      return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:       int k = (int) x;
506:       return k+1;
507:   }

509:   static public void main() {
510:      int val = f(15);
510:      System.out.println(val);
511:   }
```

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

The call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:        int k = (int) x;
506:        return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

Call Stack

Keeps track of the local variables and return location for the current function

Allows program to jump from function to function without losing track of where the program should resume

stack frame: records the information for each function call:

- local variables
- address of where to resume processing after this function is complete.

computer only needs to ***add or remove items from the very top of the stack***,

stacks are a very useful data structure for ***Last-In-First-Out (LIFO)*** processing

h
Return val to g line 501
x=56
k=56
val = 57

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

Stacks Summary

Simple and surprisingly useful data structure

First In Last Out (FILO)

Can store any number of items

User can only interact with the top of the stack:

- Push: add a new element to the top
- Pop: take off the top element
- Peek: view the top element without removing it

JUnit

JUnit

A *unit* testing framework for Java.

A *“unit”* is typically a method that we want to test in isolation

Let’s write JUnit tests for our BrowserHistory

Using JUnit

Import Test Annotation Framework

```
import org.junit.Test;
```

- Write tests using @Test annotation

```
@Test
public void testEmpty() {
    ArrayStack<String> stack = new ArrayStack<String>(10);
    assertTrue(stack.isEmpty());
}
```

Testing Guidelines

Test every method for correct outputs:

- Try simple and complex examples

Every exception and error condition should be tested too

Write test cases first, then implement

- Will make it easy to know when you are done