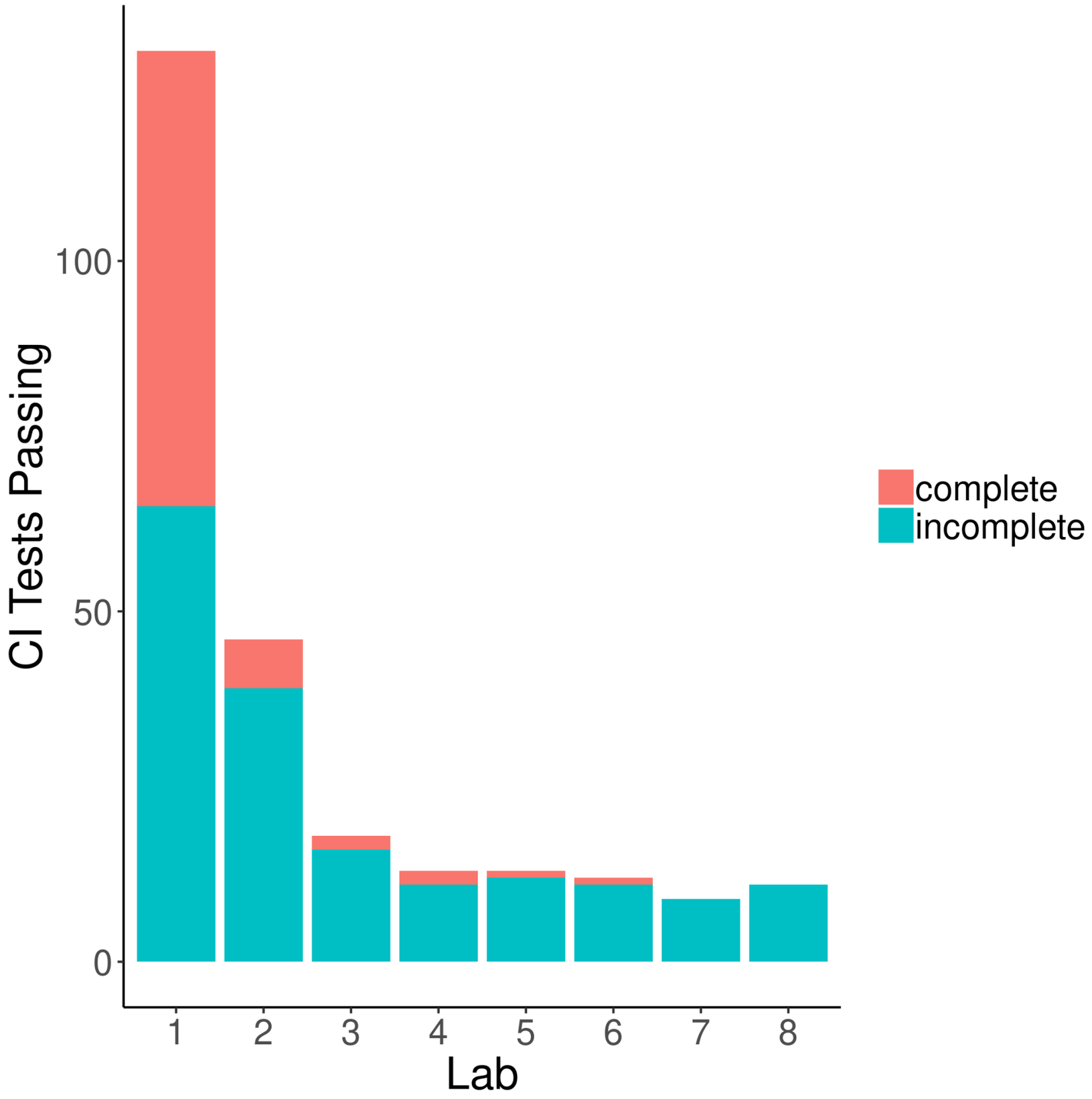

CI Tests Passing
100
50
0
1
2
3
4
5
6
7
8
Lab
complete
incomplete

Asking for formative feedback on GitLab

Software Development 3 (F27SG)

Lecture 4

Stacks

Rob Stewart

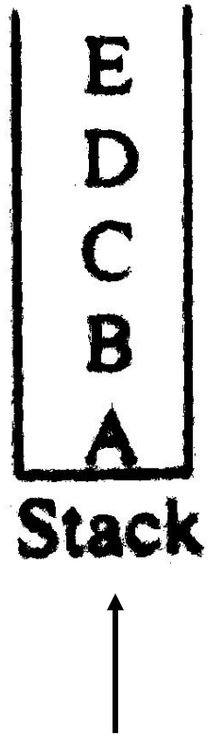
Outline

- By the end of this lecture you should
  - know what an Abstract Data Type (ADT) is
  - be able to implement and use the *Stack* ADT
  - analysis of Stacks in Big-Oh

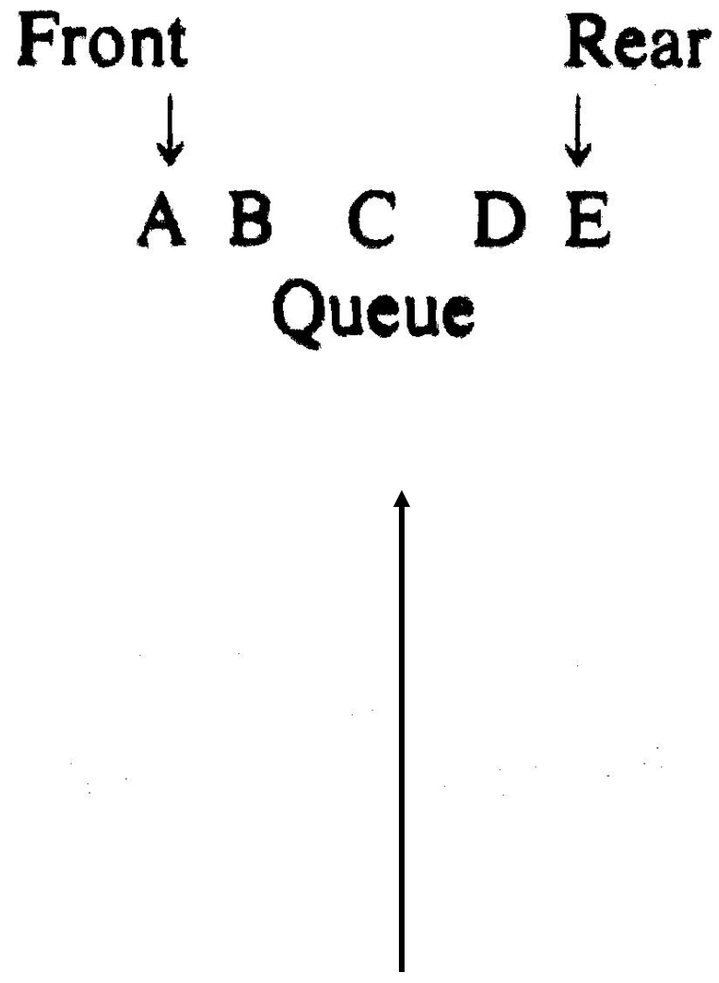
Stack Examples

- There are many examples of stack usage
  - papers in a printer
  - stack of trays in cafeteria,
  - t-shirts in a wardrobe
  - ...
- ... in *Computers*
  - undo button in a text editor
  - page visited in a web browser
  - calculating the value of expressions
  - chain of method calls in Java Virtual Machine
  - ...

Accessing a Stack



This lecture



Week 5

Fundamentals of Data Structures, E Horowitz & S Sahni, 1998.

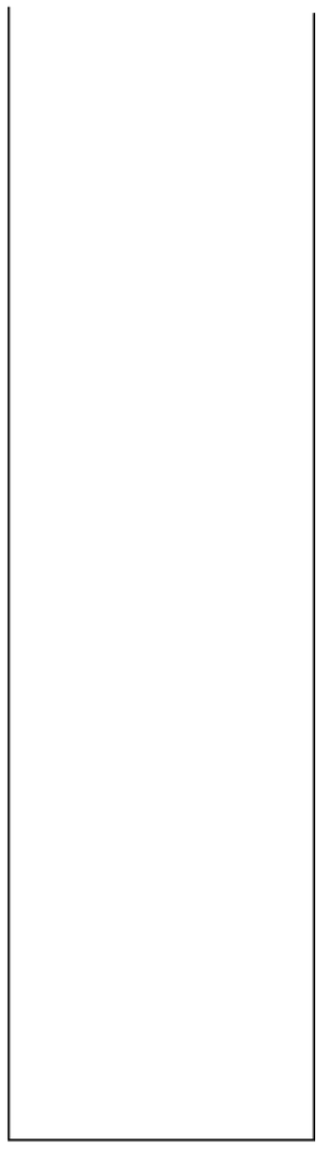
Stack Operations

Camera demo, show some operations in action:

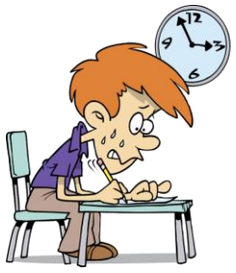
- **push**(object) - adds object to the top of the stack
- **pop**() - remove and returns element at the top
- **top**() - return element at top (also called peep())
- **size**() - return number of elements
- **IsEmpty**() - check if empty

Stack Exercise 1

Operations	return value
push(5)	
push(3)	
pop()	
push(2)	
push(8)	
pop()	
pop()	
push(9)	
push(1)	
pop()	
push(7)	
push(6)	
pop()	
pop()	
push(4)	
pop()	
pop()	



Stack



Stack Exercise 1

Initially empty stack S has performed:

- 25 push operations
- 12 top operations
- 10 pop operations where 3 of the pop generated **an exception** since the stack was empty.

What is the size of S after these operations?

www.socrative.com / `student login' / room: SD32019

Abstract Data Types (ADT)

- ADT is an abstraction of a data type specifying
  - data stored
  - operations
  - error conditions
- Example coffee shop:
  - **data:** *money, coffee, customer*
  - **operations**: *take_order(customer), make(coffee)*
  - **error conditions**: *no_money, no_coffee, ...*

The Stack ADT (1)

- A **stack** is one of the simplest ADTs
  - add/remove elements at the top

- Simplicity of operations makes a stack's behaviour
  - Reliable
  - Easy to predict

- A stack is a *LIFO* structure
  - Last In First Out
  - most recently added element is returned first

The Stack ADT (2)

- Main operations
  - **push**(object) - adds object to the top of the stack
  - **pop**() - remove and returns element at the top
- Auxiliary operations
  - **top**() - return element at top (also called peep())
  - **size**() - return number of elements
  - **isEmpty**() - check if empty
- Error conditions:
  - `pop()` empty stack
  - `top()` empty stack

Java Interfaces

- An **interface** describes methods of a class
- A list of **method declarations** without a body or data
- It has the syntax

```
public interface MyInterface { <declarations> }
```

- The keyword **implements** defines that a class C implements MyInterface:
  - all methods of MyInterface must be implemented in C

```
public class C implements MyInterface { <body>}
```

- A method can accept (as argument) or return an interface
  - meaning it accepts/returns an object implementing this interface

The Stack ADT as a Java Interface

- We can represent the Stack ADT as a Java interface
- Elements in this stack are of type **Object**
- Create a **StackException** for error conditions

```
public interface StackI {
   public int size();
   public boolean isEmpty();
   public Object top() throws StackException;
   public void push(Object element);
   public Object pop() throws StackException;
}
```

Implementing the Stack ADT

- There are many ways to implement a stack
  1. Using an **array** to store data (this lecture + lab 2)
  2. Using a **linked list** to store data (lab 4)

- Disadvantage of arrays: they have a **fixed size**, so its size is bounded by the size of the array
- Need to handle the case where array is full

```
public void push(Object element) throws StackException;
```

- This limitation is implementation specific so not part of the stack ADT
  - ***linked lists*** **to overcome this**

The StackException

- First we create a special exception **StackException** which extends **RuntimeException**
- It must include a constructor which takes a string as argument

```
public class StackException extends RuntimeException{
      public StackException(String err) {
         super(err);
      }
}
```

The Stack ADT as an Array

- Keep an array **S** of Objects, and an element **top** pointing to the top element of the stack


```
public class Stack implements Stack I{
   private int top;
   private int capacity;
   private Object[] S;
   private static int MAX = 100; // default size
```

The Stack ADT as an Array Constructors

Stack.java

- https://gitlab-student.macs.hw.ac.uk/-/ide/project/sd3/lab2/edit/master/-/src/Stack.java

The Stack ADT as an Array Constructors

- We include two constructors
  - one with the max size provided
  - one using the default size (MAX)

```
public Stack(int capacity){
    this.capacity = capacity;
    S  = new Object[capacity];
    top = -1;
}
public Stack(){
    this(MAX);
}
```

The Stack ADT as an Array Operations (1)

```
public int size(){
    return top+1;
}
public boolean isEmpty(){
    return (top < 0);
}

public void push(Object obj) throws StackException {
    if (size() == capacity)
      throw new StackException("Stack is full.");
    S[++top] = obj;
}
```

The Stack ADT as an Array Operations (2)

```
public Object top() throws StackException {
    if (isEmpty())
       throw new StackException("Stack is empty.");
    return S[top];
 }

public Object pop() throws StackException {
    if (isEmpty())
       throw new StackException("Stack is empty.");
    return S[top--];
}
```

Using a Stack

```
Stack s = new Stack();
s.push("A");
s.push("B");
String b = (String) s.pop();
```

- An **exception** will be raised when
  - **popping** an **empty** stack; or
  - **pushing** to a **full** stack
- You need to be careful to **check** that
  - a stack is not empty before popping
  - a stack is not full before pushing

The Stack ADT as an Array: *toString*

- Finally, we can update the toString() method to print out the value of the stack
  - e.g. **System.out.println** will use this method when printing a stack object

```
public String toString(){
  StringBuffer buf = new StringBuffer("[");
  if(size() > 0)
    buf.append(S[0]);
  for(int i = 1; i <= top;i++)
    buf.append(", " + S[i]);
  buf.append("]");
  return buf.toString();
}
```

Using a Stack

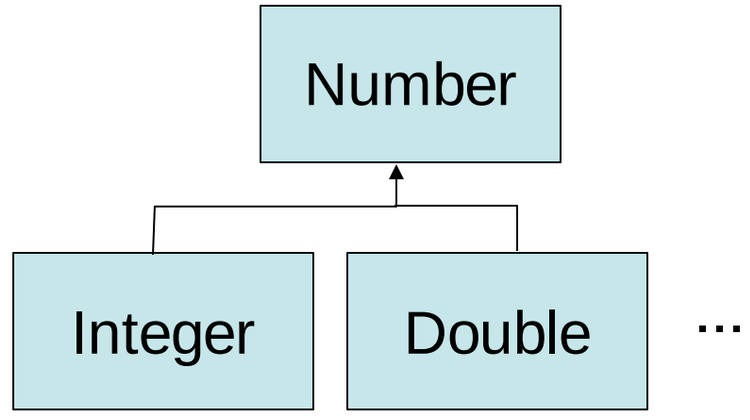
- The stack stores elements of class **Object**
- We are likely to use objects of more specialised classes
  - E.g. String, Person (from lab 1), …
- This requires a technique called **casting**

Casting

- The process of converting one type into another type is called **casting**
  - the syntax is **(type) expr**
  - however, this is often done implicitly in Java

```
double d = 3.2;
int i = (int) d;
double d2 = d / (double) i ;
```

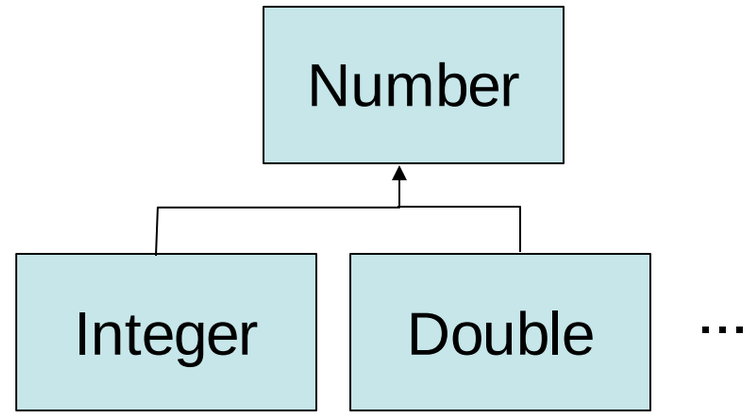
Casting



- We can also cast between objects/interfaces in a hierarchy
- **Widening** converts a type to a *wider* (more general) type
  - e.g. Integer to Number
  - it does not need explicit casting

```
Integer i = new Integer(3);
Number n = i;
```

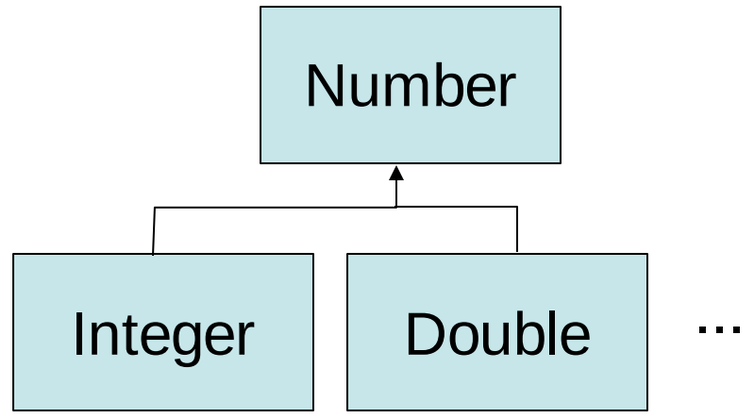
Casting



- We can also cast between objects/interfaces in a hierarchy
- **Narrowing** converts a type to a *narrower* (more specific type)
  - e.g. Number to Integer
  - it requires an explicit casting

```
Number n = new Integer(3);
i = (Integer) n;
```

Casting



In many cases you cannot cast between two types, and an **exception** will be raised

```
Number n = new Double(3.14);
i = (Integer) n;
```

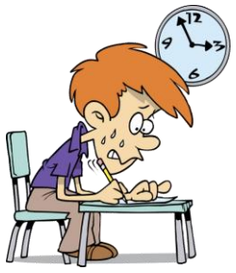
```
ClassCastException
```

Big-O for Stack Operations

- The number of primitive operations are the same for all stack operations regardless of the size
  - –e.g. consider the **size** method:

```
public int size(){
    return top+1;
}
```

  - –this will have 2 operations regardless of the stack's size
- A stack has the advantage that each operation is constant **O(1)**



Exercise

- Show that pop() is O(1) by
  - Counting the primitive
    - assume throwing an exception is a primitive operation
  - Simplify the term

```
public Object pop() throws StackException {
    if (isEmpty())
      throw new StackException("Stack is empty.");
    return S[top--];
}
```

Stacks Space Race

Summary

- In this lecture we have
  - Worked example with stack operations
  - Abstract Data Type (ADT)
  - Stack ADT: use an implementation
  - Big-Oh: the constant function: O(1)
  - Analysis of stack using big-Oh

- Attendance sheet

- Next lecture: **recursion**