Outline

- errors
- reasoning about programs
- stacks
- stack ADT
- method signatures
- array stack implementation
- linked stack implementation
- stack applications
- infix, prefix, and postfix expressions
- StringBuilder

Testing

- unless code is written correctly from the start, errors are found by testing
- but writing correct code from the start is hard, so most programmers use testing to make their code as close to correct as possible
- test routines can be included in the `main` method of any class that doesn't already have one, or in a separate test program. This is a *unit test*. This main method is a *driver program*. The driver program can also be defined separately.
- the unit test should call all the methods of the class, with as many combinations of parameters as possible
- if the writer of the test code doesn't study the code under test, this is *black box testing*
- at the very least, the goal of testing is *full coverage*: making sure that every path through the code has been used at least once, and has produced an acceptable result
- to produce full coverage, the programmer of the test program must study the code being tested: this is *white box testing*

A few common types of errors

- off-by-one (fencepost): how many fence posts are needed for a fence that is 20 feet long and has a post every 2 feet?
- not initializing data correctly. Sometimes this causes null pointer access
- using different variables as if they were one, or using one variable as if it were two variables
- assumptions that don't turn out to be true (misconceptions), not establishing and maintaining invariants
- not checking things that should be checked, e.g.

```
if (x == null)
```

strategies for testing

- print/show all method invocations and their parameters and return values (**trace**)
- write code to check that the invariants are established and maintained
- write test cases to not only provide full coverage, but also check all boundary conditions, where the result should change (make sure it changes where it should)
- some common special cases:
  - less than 0, 0, 1, greater than 1
  - first and last elements of an array, collection, linked list
  - elements and values that are null
  - desired element is not in the collection, or is in the collection more than once
  - collection has size 0, 1, or larger
- for example, when testing adding on a linked list, can test adding at the beginning of a linked list, at the end, in the middle, and adding into an empty linked list, and in both positions of a 1-element linked list. Also, adding an element that has the value `null` (is the behavior of your program defined in that case? Should it be?)
- if code to be tested needs to call a method that is not yet implemented, a *stub* of that method can do only what is needed for the test

reasoning about programs

- a **precondition** must be true before a method is called
- the code in the method is designed assuming that the precondition is true
- the caller of a method must guarantee (be sure) that the precondition holds
- a **postcondition** will be true after a method is called
- the code in the method must guarantee that the postcondition is true
- preconditions and postconditions are a little bit like a contract or any other agreement: if the caller provides the preconditions, the method will provide the postconditions
- preconditions and postconditions are documented in Javadoc
- invariants are postconditions of every method, **including** the constructors
- invariants are preconditions of every method **except** the constructors
- invariants are usually documented for the entire class rather than for each method

proof of program correctness

- given a mathematical specification
- it might be possible to prove that a program implements that specification
- specification: `add1` adds 1 and returns the result

```
private static int add1(int parameter) {
    return parameter + 1;
}
```

- seems obvious
- but fails if parameter is `Integer.MAX_VALUE`

Stacks

- stacks of dishes or trays in a cafeteria
  - maybe on a spring-loaded mechanism so only the top one is accessible
  - adding more dishes pushes down the stack, so only the new top is still accessible
- Last In First Out discipline (LIFO)
  - FIFO will be discussed at a later time, when we talk about queues

John Lehman, CC BY 3.0

Stacks in Computer Science

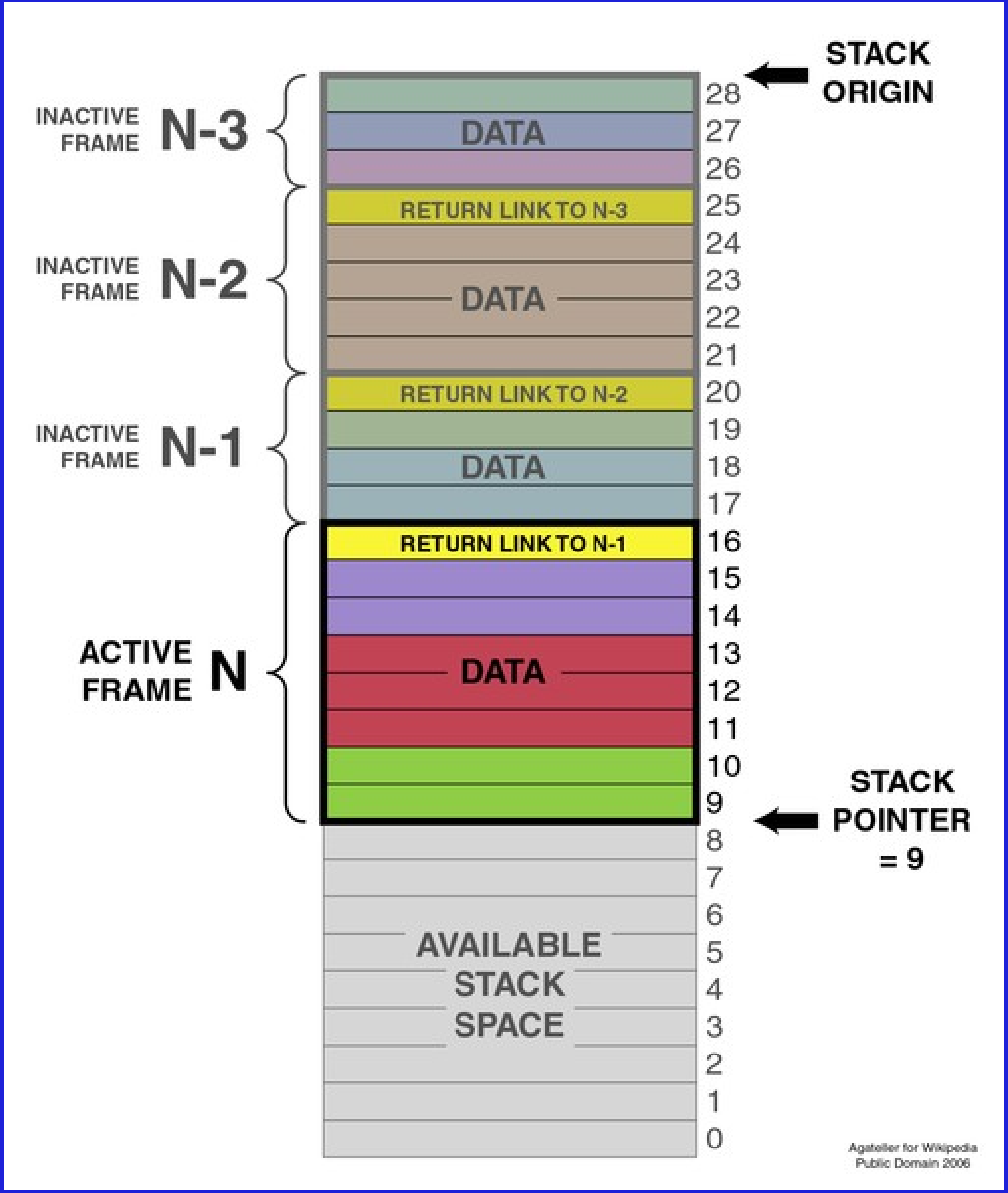
- a data structure to hold a variable number of elements
- of which only the top one is accessible at any given time
- the stack may be *empty* if it holds no elements
- the stack keeps track of the number of elements it has, as well as the individual elements
- some stack implementations set a maximum size for the stack, so the stack *overflows* if more data is added to a full stack

Stacks and other data structures

- compared to arrays and array lists:
  - an array/list does not keep track of which elements have been initialized
  - any element of an array/list is equally easy to access
  - an array has a fixed number of elements
- compared to linked lists:
  - a linked list can access any element of the list
  - even though accessing some (esp. the head) is faster than accessing others
    - random access takes O(n) time

A Stack for Method Calls (Procedure Calls)

- this stack grows downward
- each call pushes onto the stack:
  - parameters for the method being called
  - return address
  - local variables
- finite memory is available for this stack
  - too many calls overflow the stack



Stack ADT and Stack Interface

- consider a stack to store objects of some type E:
- `void push(E item)`

  adds the value to the top of the stack
- `E pop() throws EmptyStackException`

  removes the value on top of the stack and returns it (or throws the exception)
- `E peek() throws EmptyStackException`

  is like `pop`, except it doesn't remove the element
- `boolean empty()`

  reports whether the stack is empty
- only the topmost element of the stack is ever accessible
- as in every ADT, we have specified only *what* the stack does, and not *how* that is accomplished. The *how* part belongs in the implementation

Method Signatures

- can the same name be reused for different methods?
- Java: yes -- this is overloading: same name, different functionality, overloads one name with more than one function
  - programmers: yes, if they do the same thing
  - some other languages: no
- Java uses a **signature** for each method that includes the method's name, and the type/class and order (sequence) of its parameters
- each method's signature **must** be unique within a class
- example:
- `push(E)` is the signature for the push method (above)
- in an interface, this is written with the method prototype:

```
void push (E item);
```

- in a class implementation, the signature is taken from the method header:

```
/* @param  value to push onto the stack */
public void push(E value) {
```

Array stack: implementation using arrays

- even though arrays and stacks are very different, we can use an array in implementing our stack
- implementing a stack using an array is similar to implementing a list using a array
- instead of the size of the list, we keep track of where the top of the stack is: an integer we call `top`
- the code outside the class does not know and does not need to know that the stack is implemented using an array

Linked stack: implementation using linked nodes

- even though linked lists and stacks are very different, we can use linked nodes in implementing our stack
- implementing a stack using linked nodes is similar to implementing a list using linked nodes
- all operations occur at the top of the stack, which is the only node we need to keep track of
  - the top of the stack is at the head of the linked list

performance of Array Stack and Linked Stack

- given ArrayStack.java:
- in-class exercise (everyone together): what is the runtime (big O) of `empty()`, `push()`, and `pop()`?
- given LinkedStack.java:
- in-class exercise (everyone together): what is the runtime (big O) of `empty()`, `push()`, and `pop()`?

Other stack implementations, using a Java Vector or List

- any extensible data structure that has access at one end can be used to implement a stack
- this includes java Vectors and java Lists
- new data is added to or removed from the end of the Vector or ArrayList in O(1) time (when the array doesn't have to grow)
- in the code in the book (Listing 3.4, p. 164), note that the data is stored in an object of type `List<E>`, that is created as an `ArrayList<E>`: this is an example of polymorphism, and makes it easy to switch to a different kind of list
- new data is added to or removed from the end of the ArrayList, usually in O(1) time (except when the underlying array needs to grow)
- when using a LinkedList, new data is added to or removed from the front of the LinkedList in O(1) time

Stack Applications: Palindromes

- a palindrome is a string that is the same when read backwards or forwards: "radar", "level", "racecar"
- there are many algorithms for recognizing palindromes, and most are equivalent
- one such algorithm uses a stack:
    - the characters of the string are pushed onto the stack, one by one
    - as they are popped (removed) from the stack, they compared to the characters in the string
    - since they are popped in LIFO order, they are removed in reverse order
- if all the characters from the stack match the characters from the string, the string is a palindrome

Stack Applications: Matching Parentheses

- balanced parenteses: "(a (b c) [d (e)] f g)"
- unbalanced parenteses: "(a (b c {d (e)) f g]"
- algorithm to check for balanced parentheses:
  - when encountering an open parenthesis, put it on the stack
  - when encountering a closing parenthesis, remove the matching one from the top of the stack
    - or, if the top of stack does not match, or if the stack was empty, declare an error
  - at the end of the string, should have an empty stack
- if the stack is not empty at the end of the string, the parentheses are not balanced

infix expressions

- 2 + 3 * 4 has the value?
- some operators (*, /, %) have higher precedence than other operators (+, -)
- operators with the same precedence are evaluated in left-to-right order
- these expression can be evaluated using two stacks:
  - when an operand is read, it is pushed onto the operand stack
  - when reading an operator with higher precedence than the top of the operator stack, the new operator is pushed onto the operator stack
  - otherrwise, the top of the operator stack is popped and evaluated with the top two elements of the operand stack, the result is pushed onto the operand stack, and the new operator is left in the string to be read again
- at the end of the expression, operators are popped off and evaluated (popping the operands and pushing the results) until the operator stack is empty
- at this point, the operand stack should have exactly one number in it
- more interesting with more precedence levels, e.g. ^ (exponentiation), &&, ==

infix expression evaluation example

- 2 + 3 * 4 - 5 has the value?
- read 2, push it onto operand stack
- read +, push it onto operator stack
- read 3, push it onto operand stack
- read *, push it onto operator stack
- read 4, push it onto operand stack

- remaining to be read: - 5
- operand stack has: 2, 3, 4 (4 is top of stack)
- operator stack has: +, * (* is at top of stack)

infix expression evaluation example

- 2 + 3 * 4 - 5 has the value?

- operand stack has: 2, 3, 4 (4 is top of stack)
- operator stack has: +, * (* is at top of stack)
- remaining to be read: - 5

- - has lower precedence than *, so pop * from the operator stack (which now only has +), pop 4 and 3 from the operand stack, compute 3 * 4 = 12, and push 12 onto the operand stack (which now has 2, 12)
- - has the same precedence as +, so pop + from the operator stack, pop 12 and 2 from the operand stack, compute 2 + 12 = 14, and push 14 onto the operand stack (which now has 14)
- push - onto the operator stack, which now only has -
- push 5 onto the operand stack, which now has 12, 5
- at the end of the string, so pop the operator stack and (twice) the operand stack, compute 14 - 5 = 9, push 9 onto the operand stack, which now only has 9
- we're at the end of the string and the operator stack is empty, the result is 9

infix expressions in the compiler

- the Java compiler must recognize (parse) Java expressions and either:
  - evaluate any constant-valued (sub-) expressions, or
  - emit code to evaluate the expression at run-time

parenthesized infix expressions

- (2 + 3) * 4 has the value?
- when reading a left parenthesis, push it onto the operator stack
- when reading a right parenthesis, behave as at the end of the expression, until the matching left parenthesis is popped from the operator stack
- everything else is the same as the previous algorithm

integer operators

- addition (+), subtraction (-), multiplication (*) work as expected
- division (/) rounds down: 3 / 2 gives 1, 101 / 100 gives 1
- remainder (%), also known as modulo, returns the remainder from the division: 3 % 2 gives 1, 127 % 100 is 27
- multiplication, division, and modulo have higher precedence than addition and subtraction, and so are evaluated first:

    3 + 54 * 17 is 3 + (54 * 17)

- with equal-precedence operators, the expression is evaluated from left to right:

    99 - 3 - 33 / 11 / 3 is ((99 - 3) - ((33 / 11) / 3))

non-infix expressions

- in a prefix expression, the operator comes **before** the operands:

  / + 3 * 7 4 2 means (3 + 7 * 4) / 2

- in a postfix expression, the operator comes **after** the operands:

  4 2 / 3 + 1 * means (4 / 2 + 3) * 1

- in an infix expression, the operator comes **in**-between the operands
- only infix expressions need:
  - precedence
  - parentheses (to override precedence)
- in prefix and postfix expression, the position indicates which operands are used with which operators
- converting from one notation to the other can benefit from using a stack or recursion
- computing in prefix or postfix is easy when using a stack

algorithm for postfix computation

- read the next input (next character) of the string
- if the character is an operand, push it onto the stack
- if the character is an operator,
  - pop the top two elements off the stack,
  - apply the corresponding operation (the operands must be in the correct order!),
  - push the result back on the stack
- if the string is empty:
  - if the operator stack is empty and the operand stack has one element, that element is the result
  - if the operator stack is not empty, or the operand stack has 0 or multiple elements, the expression is malformed
- in-class exercise: use the above algorithm to evaluate the following expressions:

  9 7 /

  1 2 * 3 * 4 *

  3 4 * 1 2 + -

`StringBuilder`

- Java makes it easy to concatenate strings
- however, it is not particularly efficient: it involves copying all the characters of the original and the new string
- a string builder is like an array list, but for strings: it is a data structure that efficiently supports growable (extensible) strings
- a `StringBuffer` is similar, but will also work correctly in multi-threaded programs