Outline

- the AbstractCollection hierarchy
- stacks
- array stack implementation
- linked stack implementation
- stack applications
- infix, prefix, and postfix expressions

Collection Interface

- **a collection holds values of a different type E**
- **the number of values can change over time**
  - unless the collection is unmodifiable
- **collections typically have at least two constructors**
  - one with no arguments
  - one with a collection as an argument, makes a copy of the collection
  - these constructors are not defined by the interface
    - since interfaces don’t list constructors!
  - all collections in the Java standard library include these two constructors

Abstract Collection Classes

- **to create a Collection:**
  - extend `AbstractCollection` and implement `add`, `size`, and `iterator`
  - or extend `AbstractList` and implement `add`, `get`, `remove`, `set`, and `size`
  - or extend `AbstractSequentialList`, and implement `listIterator` and `size`
- **these abstract classes let you create a collection with a minimum of work**
  - this strategy is not allowed for your homeworks!
  - in-class exercise: why not?

`AbstractCollection`

- **to create a Collection, extend `AbstractCollection` and implement (override) `add`, `size`, and `iterator`**
  - no need to override the `add` method if the class is unmodifiable
- **since `Collection` has no `get` method, these three methods are sufficient for implementing a reasonable collection**
- **individual implementations may override additional methods of `Collection`**

AbstractList

- **extends `AbstractCollection`**
- **to create a `Collection`, extend `AbstractList`, and implement `add`, `get`, `remove`, `set`, and `size`**
- **this abstract list is generally intended for collections where elements are stored in a "random access" data structure such as an array**
  - `get`, `set` are typically O(1)
  - `add`, `remove` may be O(1) at the end and O(n) elsewhere
  - `set` is only needed for modifiable lists, `add` and `remove` are only needed for variable-sized lists
    - so you can have an unmodifiable, fixed-sized list by providing only `get` and `size`!!!
  - the `AbstractList` class uses `get` and `size` to implements the iterators
- **elements are stored at a specific index**

AbstractSequentialList

- **extends `AbstractList`**
- **to create a `Collection`, extend `AbstractSequentialList`, and implement `listIterator` and `size`**
- **this class is designed for sequential-access data structures such as doubly-linked lists**
- **`listIterator` is powerful enough to add and remove and find elements**
  - but the `get` operation then takes time O(n)
- **implementing the `listIterator` might be as hard as implementing the list methods directly!**
  - but if you already have a `listIterator`, the other methods are free

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

Stacks in Computer Science

- a data structure to hold a variable number of elements
- elements added and removed in Last In, First Out (LIFO) order
- only the top element is accessible at any given time
- the stack may be *empty* if it holds no elements
- the *push* operation adds a new value at the top of the stack
- the *pop* operation removes and returns the value at the top of the stack
- some stack implementations set a maximum size for the stack, so the stack *overflows* if more data is added to a full stack

Stacks compared to 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 in a linked list takes O(n) time, in a stack random access is forbidden

Stack Interface

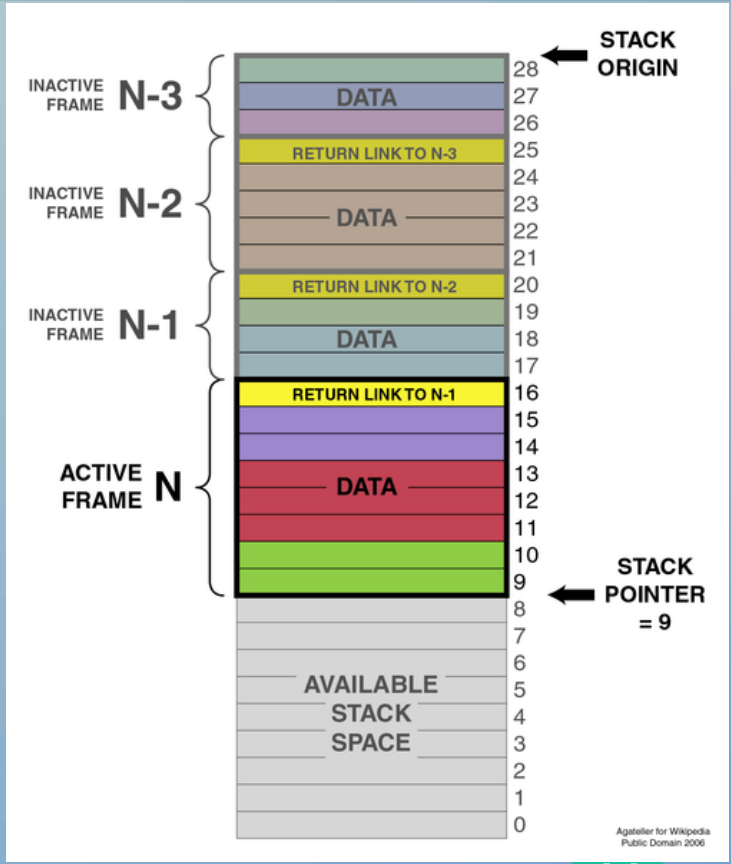
- **Last In First Out discipline (LIFO)**
  - the last value pushed onto the stack will be the first value popped

```
public interface Stack<E> {
  void push(E item);
  // E push(E item) in standard library
  E pop()
     throws EmptyStackException;
  boolean empty();
}
```

John Lehman, CC BY 3.0

the System stack:
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 where the computation starts again after returning
  - local variables
- **this stack only has finite space**
  - too many calls overflow the stack



Array stack:
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 an 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:
stack implementation using linked nodes

- **even though linked lists and stacks are very different, we can use linked nodes to implement 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 ArrayStack and LinkedStack

- 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**
  - also an example of using as type the name of an interface
- **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 (and take O(n) time, where n is the length of the string)**
- **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 are 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
  - otherwise, 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 (create) 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 (especially 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 + -