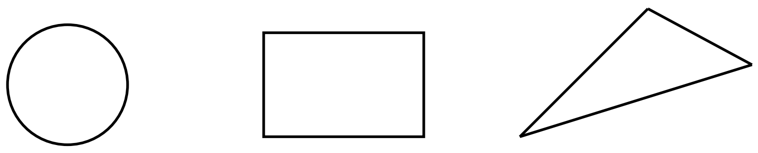
CSE 143
Lecture 9

Interfaces; Stacks and Queues

slides created by Marty Stepp
http://www.cs.washington.edu/143/

Related classes

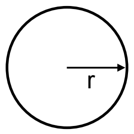
- Consider the task of writing classes to represent shapes such as `Circle`, `Rectangle`, and `Triangle`.



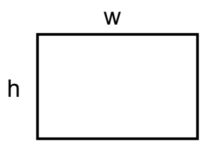
- Certain operations are common to all shapes:
  - perimeter
  - area

Shape area and perimeter

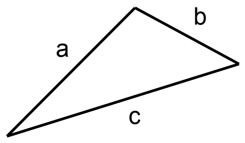
- Circle:
  - area = πr^2
  - perimeter = $2 \pi r$



- Rectangle:
  - area = $w h$
  - perimeter = $2w + 2h$



- Triangle:
  - area = $\sqrt{(s (s - a) (s - b) (s - c))}$
    - where $s = ½ (a + b + c)$
  - perimeter = $a + b + c$



Printing shape information

- We'd like our client code to be able to print the area and perimeter of our shapes.
  - Of course, we would use methods...

```
public static void printCircleInfo(Circle c) {
   System.out.println("area = " + c.area());
   System.out.println("perimeter = " + c.perimeter());
}

public static void printTriangleInfo(Triangle t) {
   System.out.println("area = " + t.area());
   System.out.println("perimeter = " + t.perimeter());
}

public static void printRectangleInfo(Rectangle r) {
   System.out.println("area = " + r.area());
   System.out.println("perimeter = " + r.perimeter());
}
```

- Redundancy—ewwwwwww!!!! What can we do about this?

Printing shape information

- What about having shapes inherit from a superclass `Shape` that has the `area` and `perimeter` methods so we can have *polymorphic* methods?

```
public class Rectangle extends Shape { ... }
public class Triangle extends Shape { ... }
public class Circle extends Shape { ... }

public static void printShapeInfo(Shape s) {
   System.out.println("area = " + s.area());
   System.out.println("perimeter = " + s.perimeter());
}
```

- What is wrong with this solution?
  - What does the `Shape` class look like?

5

Inheritance doesn't apply

```
public class Shape {
    public double area() {
        // what goes here?
    }

    public double perimeter() {
        // what goes here?
    }
}
```

- This doesn't work—different shapes compute the area and perimeter differently!

6

Interfaces (9.5)

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.
      This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface.
      This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

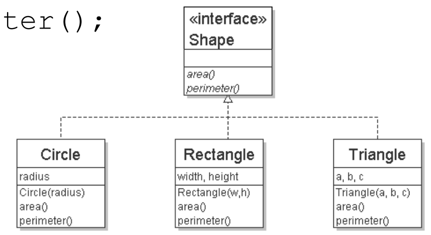
Example:

```
public interface Vehicle {
    public int getSpeed();
    public void setDirection(int direction);
}
```

Shape interface

```
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}
```

- Saved as `Shape.java`



- **abstract method**: A header without an implementation.
  - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.

Implementing an interface

```
public class name implements interface {
    ...
}
```

- A class can declare that it "implements" an interface.
  - The class promises to contain each method in that interface. (Otherwise it will fail to compile.)
  - Example:

```
public class Bicycle implements Vehicle {
    ...
}
```

Interface requirements

```
public class Banana implements Shape {
    // no methods!
}
```

- If we write a class that claims to be a `Shape` but doesn't implement `area` and `perimeter` methods, it will not compile.

```
Banana.java:1: Banana is not abstract and does
not override abstract method area() in Shape
public class Banana implements Shape {
                ^
```

Polymorphism

- Interfaces benefit the *client code* author the most.
  - they allow **polymorphism**
    (the same code can work with different types of objects)

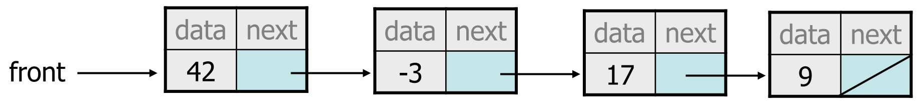
```
public static void printInfo(Shape s) {
    System.out.println("area: " + s.area());
    System.out.println("perimeter: " + s.perimeter());
}
...
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```

Linked vs. array lists

- We have implemented two collection classes:
  - `ArrayIntList`

index	0	1	2	3
value	42	-3	17	9

  - `LinkedIntList`



  - They have similar behavior, implemented in different ways. We should be able to treat them the same way in client code.

13

An `IntList` interface

```
// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}


public class ArrayIntList implements IntList { ...
public class LinkedIntList implements IntList { ...
```

14

Redundant client code

```
public class ListClient {
    public static void main(String[] args) {
        ArrayIntList list1 = new ArrayIntList();
        list1.add(18);
        list1.add(27);
        list1.add(93);
        System.out.println(list1);
        list1.remove(1);
        System.out.println(list1);

        LinkedIntList list2 = new LinkedIntList();
        list2.add(18);
        list2.add(27);
        list2.add(93);
        System.out.println(list2);
        list2.remove(1);
        System.out.println(list2);
    }
}
```

15

Client code w/ interface

```
public class ListClient {
    public static void main(String[] args) {
        IntList list1 = new ArrayIntList();
        process(list1);

        IntList list2 = new LinkedIntList();
        process(list2);
    }

    public static void process(IntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1);
        System.out.println(list);
    }
}
```

16

ADTs as interfaces (11.1)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:

– `ArrayList` and `LinkedList`	implement `List`
– `HashSet` and `TreeSet`	implement `Set`
– `LinkedList` , `ArrayDeque`, etc.	implement `Queue`

17

Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<String>();
```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type so they can be used as widely as possible:

```
public void stutter(List<String> list) {
    ...
}
```

18

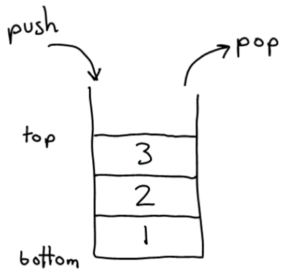
Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?
- Answer: Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end; `LinkedList` is faster for adding/removing at the front/middle.
  - `HashSet` can search a huge data set for a value in short time; `TreeSet` is slower but keeps the set of data in a sorted order.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work with minimal changes (only when the object is created).

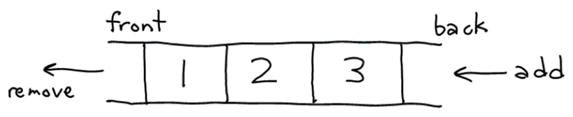
19

Stacks and queues

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.
- We will examine two specialty collections:
  - **stack**: Retrieves elements in the reverse of the order they were added.
  - **queue**: Retrieves elements in the same order they were added.



stack



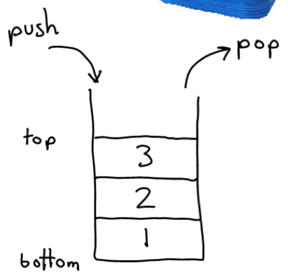
queue

20

Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.

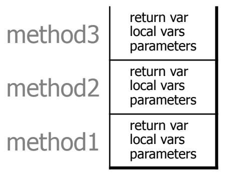
- basic stack operations:
  - **push**: Add an element to the top
  - **pop**: Remove and return the top element



21

Stacks in computer science

- Programming languages:
  - method calls are placed onto a stack
    (call=push, return=pop)



- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces `{ }` and other operators match
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations (Edit→Undo)

22

Interface `Stack`

`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<Integer> s = new ArrayStack<Integer>();
s.push(42);
s.push(-3);
s.push(17);                       // bottom [42, -3, 17] top
System.out.println(s.pop()); // 17
```

- NOTE: This `Stack` *interface* differs from the `Stack` *class* in `java.util`. We are using a custom interface, because Java messed up and did not include an interface for stacks. `ArrayStack` is a custom class (available from course web page).

Stack limitations/idioms

- Remember: You cannot loop over a stack in the usual way.

```
Stack<Integer> s = new ArrayStack<Integer>();
...
~~for (int i = 0; i < s.size(); i++) {~~
~~    do something with s.get(i);~~
~~}~~
```

- Instead, you must pull contents out of the stack to view them.
  - common idiom: Removing each element until the stack is empty.

```
while (!s.isEmpty()) {
    do something with s.pop();
}
```

Exercise

- Consider an input file of exam scores in reverse ABC order:

```
Ziggy  87
Wendy  84
Tom    52
Jerry  95
...
```

- Write code to print the exam scores in ABC order using a stack.

25

Solution

```
import java.io.*;
import java.util.*;

public class StackExercise {
   public static void main(String[] args) throws FileNotFoundException {
      Scanner input = new Scanner(new File("scores.txt"));
      Stack<String> s = new ArrayStack<String>();

      while (input.hasNextLine()) {
         s.push(input.nextLine());
      }

      while (!s.isEmpty()) {
         System.out.println(s.pop());
      }
   }
}
```

- What if we wanted to do more processing on the exam scores?

26

What happened to my stack?

- Suppose we're asked to write a method `max` that accepts a Stack of integers and returns the largest integer in the stack.
  - The following solution is seemingly correct:

```
// Precondition: s.size() > 0
public static int max(Stack<Integer> s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = Math.max(maxValue, next);
    }
    return maxValue;
}
```

  - The algorithm is correct, but what is wrong with the code?

27

What happened to my stack?

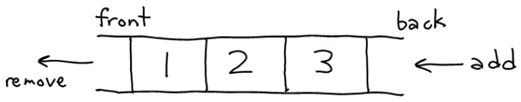
- The code destroys the stack in figuring out its answer!
  - To fix this, you must save and restore the stack's contents:

```
public static int max(Stack<Integer> s) {
    Stack<Integer> backup = new ArrayStack<Integer>();
    int maxValue = s.pop();
    backup.push(maxValue);

    while (!s.isEmpty()) {
        int next = s.pop();
        backup.push(next);
        maxValue = Math.max(maxValue, next);
    }

    while (!backup.isEmpty()) {
        s.push(backup.pop());
    }
    return maxValue;
}
```

28

Queues

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:
  - **enqueue** (add): Add an element to the back
  - **dequeue** (remove): Remove the front element

29

Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)
  - remembering which Goomba is supposed to attack Mario next

30

Programming with Queues

`enqueue(`**value**`)`	places given value at back of queue
`dequeue()`	removes value from front of queue and returns it; throws a `IllegalStateException` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedQueue<Integer>();
q.enqueue(42);
q.enqueue(-3);
q.enqueue(17);          // front [42, -3, 17] back]
System.out.println(q.dequeue());    // 42
```

- NOTE: We will also be using a custom `Queue` interface (which is a simplified version of Java's `Queue` interface).

31

Queue Mystery

- Let's try printing the contents of a queue:

```
public class QueueMystery {
   public static void main(String[] args) {
      Queue<Integer> q = new LinkedQueue<Integer>();
      for (int i = 0; i < 10; i++) {
         q.enqueue(i);
      }

      for (int i = 0; i < q.size(); i++) {
         System.out.println(i + ": " + q.dequeue());
      }
   }
}
```

- What is the output?

32

Queue idioms

- As with stacks, must pull contents out of queue to view them.

```
while (!q.isEmpty()) {
    do something with q.dequeue();
}
```

  - another idiom: Examining each element exactly once.

```
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.dequeue();
    (including possibly re-adding it to the queue)
}
```

33

Mixing stacks and queues

- We often mix stacks and queues to achieve certain effects.
  - Example: Reverse the order of the elements of a queue.

```
Queue<Integer> q = new LinkedQueue<Integer>();
q.enqueue(1);
q.enqueue(2);
q.enqueue(3)                   // [1, 2, 3]

Stack<Integer> s = new ArrayStack<Integer>();
while (!q.isEmpty()) {         // Q -> S
    s.push(q.dequeue());
}
while (!s.isEmpty()) {         // S -> Q
    q.enqueue(s.pop());
}
System.out.println(q);         // [3, 2, 1]
```

34

Exercises

- Write a method `stutter` that accepts a queue of integers as a parameter and replaces every element of the queue with two copies of that element.
  - `front [1, 2, 3] back`
    becomes
    `front [1, 1, 2, 2, 3, 3] back`

- Write a method `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.
  - `front [a, b, c] back`
    becomes
    `front [a, b, c, c, b, a] back`

35

Solutions

```
public static void stutter(Queue<Integer> q) {
   int size = q.size();
   for (int i = 0; i < size; i++) {
      int value = q.dequeue();
      q.enqueue(value);
      q.enqueue(value);
   }
}

public static void mirror(Queue<String> q) {
   Stack<String> s = new ArrayStack<String>();

   int size = q.size();
   for (int i = 0; i < size; i++) {
      String value = q.dequeue();
      q.enqueue(value);
      s.push(value);
   }

   while (!s.isEmpty()) {
      q.enqueue(s.pop());
   }
}
```

36