CSE 373

Implementing a Stack

Reading: Weiss Ch. 3; 3.6; 1.5

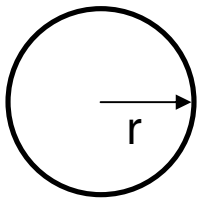
slides created by Marty Stepp
http://www.cs.washington.edu/373/

Related classes

Consider classes for shapes with common features:

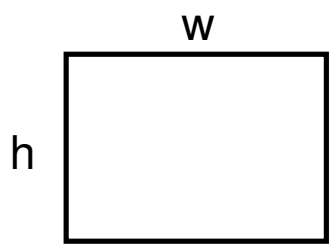
- Circle (defined by radius *r*):

  area = πr^2, perimeter = $2 \pi r$



- Rectangle (defined by width *w* and height *h*):

  area = $w h$, perimeter = $2w + 2h$

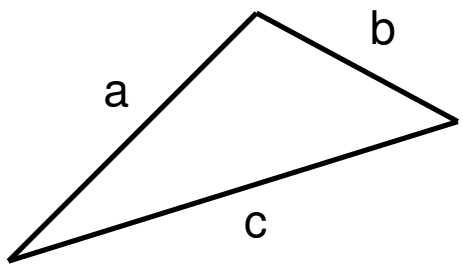


- Triangle (defined by side lengths *a*, *b*, and *c*)

  area = $\sqrt{(s\,(s-a)\,(s-b)\,(s-c))}$

  where $s = \frac{1}{2}\,(a + b + c)$,

  perimeter = $a + b + c$



  - Every shape has these, but each computes them differently.

Interfaces

- **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 {
    type name(type name, ..., type name);
    type name(type name, ..., type name);
    ...
}
```

Example:

```
// Features common to all shapes.
public interface Shape {
    double area();
    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.
  - Then the class *must* contain each method in that interface.

```
public class Rectangle implements Shape {
    public double area() { return w * h; }
    …
}
```

(Otherwise it will fail to compile.)

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

Polymorphism

- **polymorphism**: The *client* of your classes can use the same code to work with different types of objects.

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

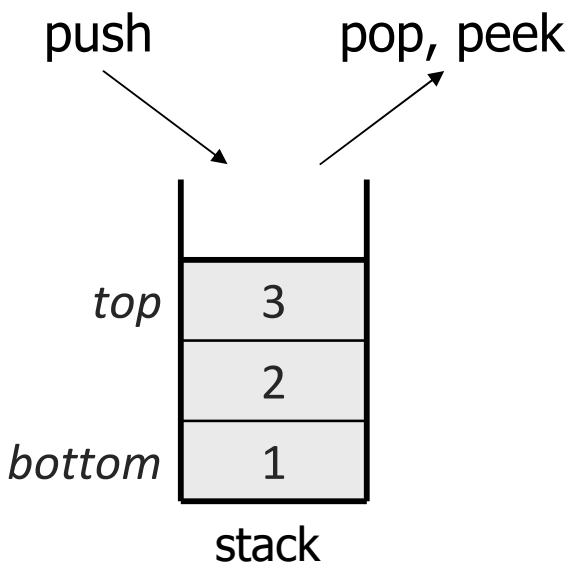
Java ADT interfaces

- Java describes its collection ADTs as interfaces:
  - `public interface Collection<E>`
  - `public interface List<E>`
  - `public interface Map<K, V>`
  - `public class ArrayList<E> implements List<E>`
  - `public class LinkedList<E> implements List<E>`
  - `public class HashMap<K, V> implements Map<K, V>`

- This means you can write one piece of code that can work with any `List`, or any `Set`, or any `Collection`, ...
  - `public static int max(List<Integer> list) { ...`
  - `private Set<String> names;`
  - `public Map<String, Integer> getScores() { ...`

Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top").



- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.

Recall: Java's Stack class

`Stack<E>()`	constructs a new stack with elements of type **E**
`push(value)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

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

- `Stack` does not use an ADT interface; it is poorly designed.
- If we were to re-implement `Stack` properly, how would it look?

Int Stack ADT interface

- Let's write our own implementation of a stack.
  - To simplify the problem, we only store `int`s in our stack for now.
  - As is (usually) done in the Java Collection Framework, we will define stacks as an ADT by creating a stack interface.

```
public interface IntStack {
    void clear();
    boolean isEmpty();
    int peek();
    int pop();
    void push(int value);
    int size();
}
```

Implementing w/ array

```
public class ArrayIntStack implements IntStack {
    private int[] elements;
    private int size;
    ...
}
```

- A stack can be implemented efficiently with an *unfilled* array.
  - An array plus a `size` field to remember the indexes used.

```
s.push(26);    // client code
s.push(-9);
s.push(14);
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	26	-9	14	0	0	0	0	0	0	0
size	3									

Implementing push

- How do we push an element onto the end of a stack?

```
public void push(int value) {   // just put the element
    elements[size] = value;     // in the last slot,
    size++;                     // and increase size
}
```

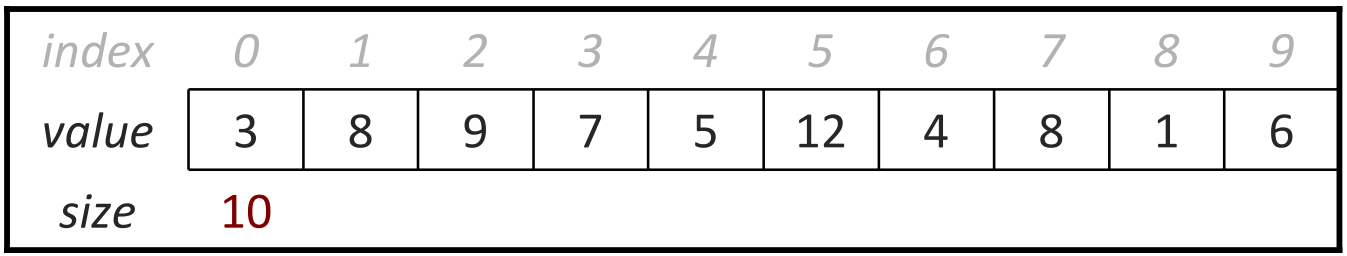
index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	0	0	0	0
size	6									

```
s.push(42);  // client code
```

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	**42**	0	0	0
size	**7**									

Running out of space

- What to do if client needs to add more than 10 elements?



  - `s.push(15);  // add an 11th element`

- Resize the array if necessary:

```
public void push(int value) {
    if (size == elements.length) {
        elements = Arrays.copyOf(elements, 2*size);
    }
    elements[size] = value;
    size++;
}
```

The Arrays class

- Class `Arrays` in `java.util` has many useful array methods:

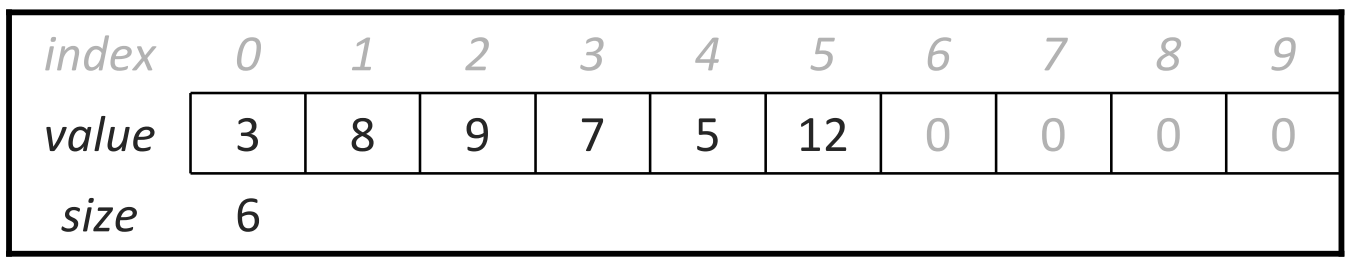
Method name	Description
`binarySearch(`**array, value**`)` or `(`**array, start, end, value**`)`	returns the index of the given value in a *sorted* array (or < 0 if not found)
`copyOf(`**array, length**`)`	returns a new resized copy of an array
`equals(`**array1, array2**`)`	returns `true` if the two arrays contain same elements in the same order
`fill(`**array, value**`)`	sets every element to the given value
`sort(`**array**`)`	arranges the elements into sorted order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, -25, 17]"`

  - Syntax: `Arrays.`**methodName**`(`**parameters**`)`

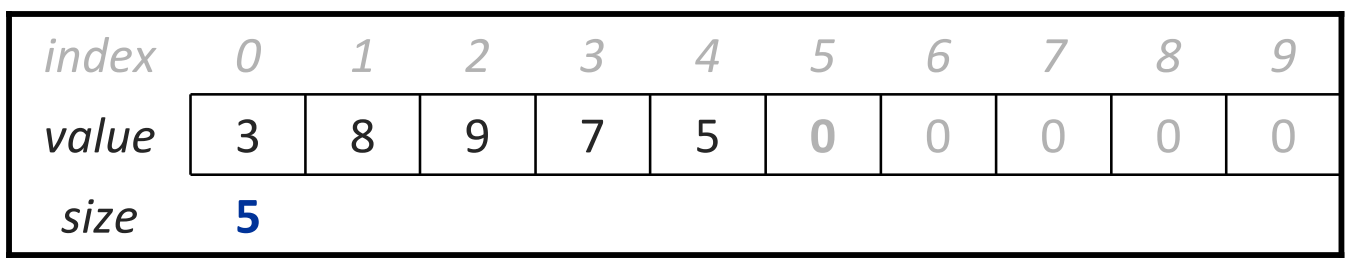
Implementing pop

- How do we pop an element off the end of a stack?

```
public int pop() {
    int top = elements[size - 1];
    elements[size - 1] = 0;    // remove last element
    size--;                    // and decrease size
    return top;
}
```



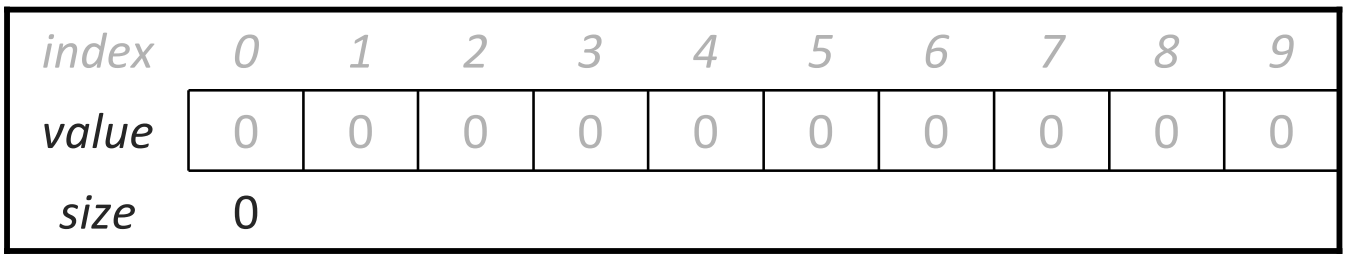

```
s.pop();    // client code;  returns 12
```



Popping an empy stack

- What if the client tries to pop from an empty stack?

```
IntStack s = new ArrayIntStack();
s.pop();   // client code
```



  - What "should" happen?
  - What is the right action for the stack to take in this case?
  - What do Java's collections do in cases like this one?

Throwing exceptions

```
throw new ExceptionType();
throw new ExceptionType("message");
```

- Generates an exception that will crash the program, unless the client has code to handle ("catch") the exception.

- Common exception types:
  - ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException, EmptyStackException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException

- Why would anyone ever *want* a program to crash?

Commenting exceptions

- If your method potentially throws any exceptions, you should comment them in its header; explain what exception and why.

```
/**
 * Removes and returns the top element of the stack.
 * Throws an EmptyStackException if stack is empty.
 */
public int pop() {
    if (size == 0) {
        throw new EmptyStackException();
    }
    int top = elements[size - 1];
    elements[size - 1] = 0;   // remove last element
    size--;                   // and decrease size
    return top;
}
```

Other methods

- Let's implement the following methods in our stack class:
  - `peek()`
    Returns the element on top of the stack, without removing it.
  - `size()`
    Returns the number of elements in the stack.
  - `isEmpty()`
    Returns `true` if the stack contains no elements; else `false`.
    (Why write this if we already have the `size` method?)
  - `clear()`
    Removes all elements from the stack.
  - `toString()`
    Returns a string representation of the stack's elements.

Type parameters (generics)

```
List<Type> name = new ArrayList<Type>();
```

- Recall: When constructing Java collections, you specify the type of elements it will contain between < and >.
  - We say that the `List` accepts a **type parameter**, or that it is a **generic** class.

```
List<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");

List<Integer> scores = new ArrayList<Integer>();
scores.add(17);
scores.add(12);
```

Implementing a generic class

```
// a parameterized (generic) class
public class name<TypeParam> {
    ...
}
```

- By putting a **TypeParam** in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - You can require multiple type parameters separated by commas.
  - Don't write a specific type like `String`; write a type variable like `T` or `E`.
    - The client gives a value to that type variable on object construction.

- The rest of your class's code can refer to that type by name.
- *Exercise:* Convert our stack interface/class to use generics.

Stack<E> ADT interface

- Let's modify our stack interface to be generic.
  - Anywhere that we expected an `int` element value, change it to `E`.
  - Not *all* occurrences of `int` change to `E`; only ones about elements.
  - We will also need to modify our `ArrayIntStack` class...

```
public interface Stack<E> {
    void clear();
    boolean isEmpty();
    E peek();
    E pop();
    void push(E value);
    int size();
}
```

Generic type limitations

```
public class Foo<T> {
    private T myField;                          // ok

    public void method1(T param) {
        myField = param;                        // ok
        T temp = new T();                       // error
        T[] array = new T[10];                  // error
    }
}
```

- If my class accepts type parameter `T`, what is a `T`? What can a `T` do?
  - Essentially nothing; think of a `T` as just any general `Object`.
  - You *can* create variables, fields, parameters, and returns of type `T`.
  - *Can't* call any type-specific methods on it, like length, toUpperCase, sort...
  - *Can't* construct a new object of type `T`.
  - *Can't* directly construct a new array of T objects (`T[]`).
    - *(But a work-around is to construct a new `Object[]` and cast to `T[]`...)*