Interfaces

- Stacks have push, pop, and isEmpty methods. There are lots of implementations — array, ArrayList, LinkedList, among others. We can describe how all of them work using an *interface*.

Interfaces

- All interface methods are automatically public.
- Two ways to think of an interface:
  - A guarantee to the client code that any class that implements the interface definitely has methods with these headers (and maybe other methods too).
  - An obligation of the implementer, who must write these methods.

A Stack implementation

```
/** A Stack with fixed capacity. */
public class ArrayStack implements Stack {
  /** The index of the top element in this Stack.  Also the number. */
  private int top;
  /** contents[0 .. top-1] contains the elements in this Stack. */
  private Object[] contents;
  /** An ArrayStack with capacity for n elements. */
  public ArrayStack(int n) {
    contents = new Object[n];
  }
  /** Add o to the top. (Ignore that we might overflow.) */
  public void push(Object o) {
    contents[top++] = o;
  }
  /** Remove and return the top element of this Stack. */
  public Object pop() {
    return contents[--top]; // What if top is 0?
  }
  /** Return true iff this Stack is empty. */
  public boolean isEmpty() {
    return top == 0;
  }
}
```

Using a Stack

- You can't create instances of interfaces. This is broken:

```
Stack s = new Stack(15);
```

- But you can write methods that use an interface:

```
/**
 * Fill a stack with the integers 0 to n - 1 (inclusive),
 * with n - 1 at the top.
 * @param the Stack to fill
 * @param n the number of integers to put into the stack
 */
public static void fill(Stack s, int n) {
  for (int i = 0; i != n; i++) {
    s.push(new Integer(i));
  }
}
```

- That function will work with *any* class that implements Stack. You should think of it as a service: it does work for anyone who needs their Stack filled with integers.

Queues (as an intro to generics)

- Queue ops: enqueue, head, dequeue, size. Let's also decide that all items in a queue must be the same type.

```
/** A queue where all items must be of type T. */
public interface Queue<T> {
  /** Append o to me. */
  void enqueue(T o);
  /**
   * Return my front item.
   * Precondition: size() != 0.
   */
  T head();
  /**
   * Remove and return my front item.
   * Precondition: size() != 0.
   */
  T dequeue();
  /** Return my number of items. */
  int size();
}
```

Queues (as an intro to generics)

```
/** A queue where all items must be of type T. */
public class LinkedListQueue<T> implements Queue<T> {
  /** The items in me.  Head is index 0, tail is index size() - 1. */
  private LinkedList<T> contents = new LinkedList<T>();
  @Override
  public void enqueue(T item) {
    contents.add(item);
  }
  @Override
  public T head() {
    return contents.get(0);
  }
  @Override
  public T dequeue() {
    return contents.removeFirst();
  }
  @Override
  public int size() {
    return contents.size();
  }
}
```

Queues (as an intro to generics)

```
public class QueueDemo {
  public static void fill(Queue<Integer> queue, int num) {
    for (int i = 0; i != num; i++) {
      queue.enqueue(i);
    }
  }

  public static void main(String[] args) {
    // Here is where we decide which Queue implementation to use.
    Queue<Integer> queue = new LinkedListQueue<>();
    fill(queue, 10);
    System.out.println(queue);
  }
}
```

Generics

- "class Foo<T>" introduces a class with a type parameter T.
- "<T extends Bar>" introduces a type parameter that is required to be a descendant of the class Bar — with Bar itself a possibility.
  In a type parameter, "extends" is also used to mean "implements".
- "<? extends Bar>" is a type parameter that can be any class that extends Bar. We'll never refer to this type, so we don't give it a name.
- "<? super Bar>" is a type parameter that can be any ancestor of Bar.