Interfaces & Collections

CSC207 Summer 2018

Abstract Classes

A class declared `abstract`:

may or may not have abstract methods (methods declared without implementation)

can’t be initiated, but can be extended

A child class may implement some or all of the inherited abstract methods.

If not all, it must be declared `abstract`.

If all, it’s not abstract and so can be instantiated.

The Programming Interface

The "user" for almost all code is a programmer. That user wants to know:

... what kinds of object your class represents

... what actions it can take (methods)

... what properties your object has (getter methods)

... what guarantees your methods and objects require and offer

... how they fail and react to failure

... what is returned and what errors are raised

Interfaces

An interface is (usually) a class with no implementation.
It has just the method signatures and return types.
It guarantees capabilities.
It is like a contract between groups of programmers.

Example: `java.util.List`
"To be a List, here are the methods you must support."

All interface methods are automatically public.

A class can be declared to `implement` an interface.
This means it defines a body for every method.
A class can implement 0 or more interfaces (but may extend only 0 or 1 classes).

An interface may extend another interface.

Interfaces vs abstract classes

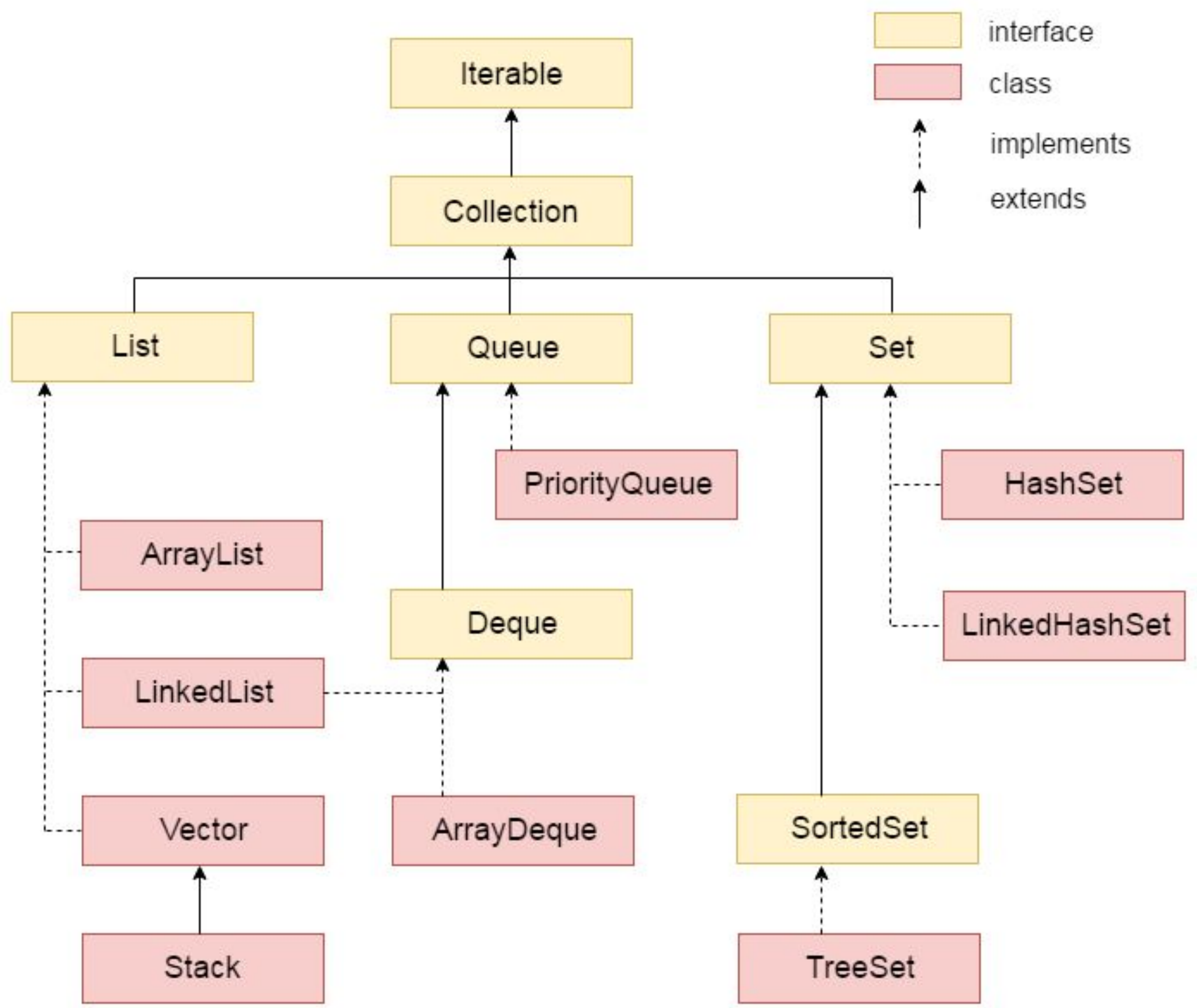
- Neither can be instantiated
- Abstract classes can have fields

See more details below:

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Collections in Java

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.

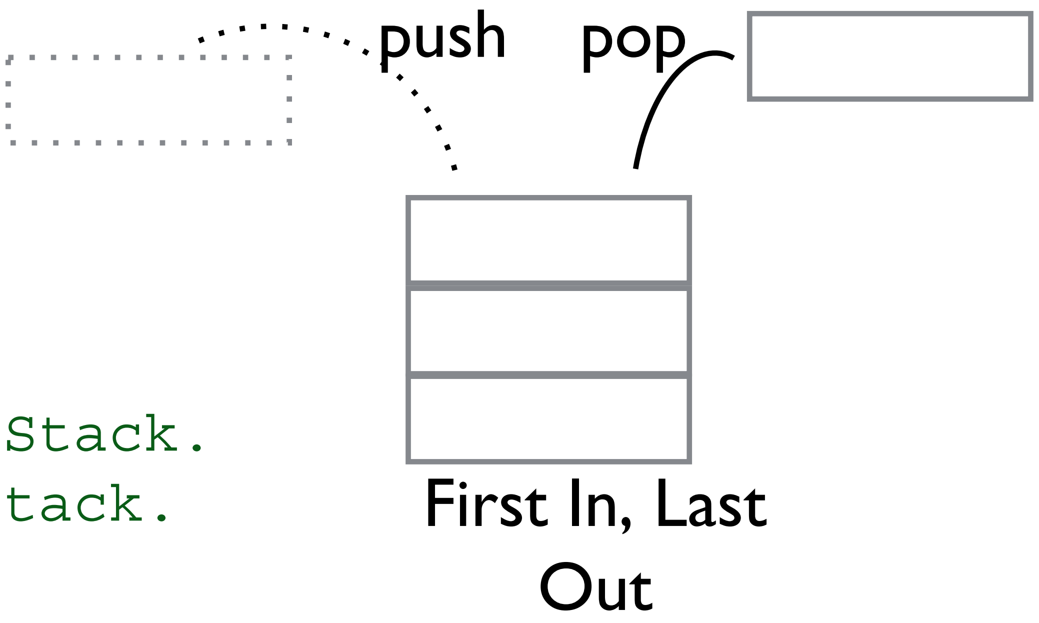


Reference: https://www.javatpoint.com/collections-in-java

Interfaces - Stack Example

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*.

```
/** A LIFO data structure. */
public interface Stack {
  /** Add o to the top of this Stack.
  * @param o The object to be pushed.
   */
  void push(Object o);
  /** Remove and return the top item of this Stack.
  * @return the former top item of the this Stack.
   */
  Object pop();
  /** Return the top item of this Stack.
   * @return the top item of the this Stack.
   */
  Object top();
  /** Return whether this Stack is empty. */
  boolean isEmpty();
}
```



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:

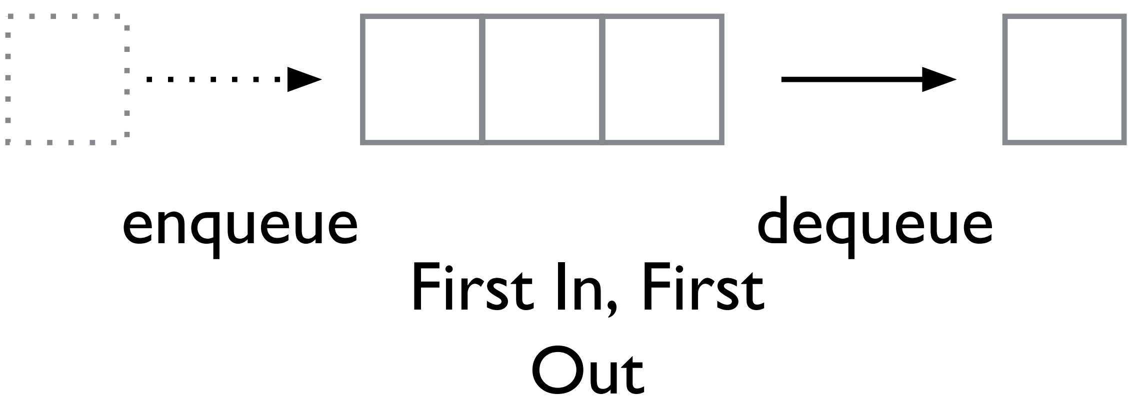
```
/**
 * Fill a stack with the integers 0 to n - 1 (inclusive),
 * with n - 1 at the top.
 * @param the Stack to fill
 * @param n the number of integers to put into the stack
 */
public static void fill(Stack s, int n) {
  for (int i = 0; i != n; i++) {
    s.push(new Integer(i));
  }
}
```

That function will work with *any* class that implements Stack.

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: naming conventions

The Java Language Specification recommends these conventions for the names of type variables:

very short, preferably a single character

but evocative

all uppercase to distinguish them from class and interface names

Specific suggestions:

Maps:	K, V
Exceptions:	X
Nothing particular:	T (or S, T, U or T1, T2, T3 for several)

More on this in future lectures