**Pre-screen questions**

1. Explain what 'path to root' means in the context of garbage collection. What are roots?

To determine an object's strength of reachability, the garbage collector starts at the root set and traces all the paths to objects on the heap. "roots" are effectively include any objects in the older generation that point to an object in the younger collection. GC Roots normally include references from application and JVM-internal static fields, and from thread stack-frames, all of which effectively point to the application’s reachable object graphs.

2. Write code for a simple implementation of HashMap/Hashtable

**import** java.awt.\*;

**import** java.util.ArrayList;

**import** java.util.LinkedList;

**public** **class** Hash<K, V> {

ArrayList<LinkedList<KV<K, V>>> buckets = **new** ArrayList<>();

**public** Hash() {

**for** (**int** a = 0; a < 256; a++)

buckets.add(**new** LinkedList<KV<K, V>>());

}

**public** **static** **void** main(String[] args) {

Hash<String, Point> hash = **new** Hash<>();

hash.put("bob", **new** Point(4, 6));

hash.put("blab", **new** Point(10, 12));

System.***out***.println(hash.get("bob"));

System.***out***.println(hash.get("blab"));

}

**private** **void** put(K key, V value) {

buckets.get(key.hashCode() & 0xFF).add(**new** KV<>(key, value));

}

**public** V get(K key) {

**for** (KV<K, V> kv: buckets.get(key.hashCode() & 0xFF))

**if** (kv.key.equals(key))

**return** kv.value;

**throw** **new** IllegalStateException();

}

**static** **class** KV<K, V> {

**private** **final** K key;

**private** **final** V value;

**public** KV(K key, V value) {

**this**.key = key;

**this**.value = value;

}

}

}

3. Write a short program to illustrate the concept of deadlock

**public** **class** Deadlock {

**public** **static** **void** main(String[] args) {

**final** Object lock1 = **new** Object();

**final** Object lock2 = **new** Object();

*locker*(lock1, lock2).start();

*locker*(lock2, lock1).start();

}

**static** Thread locker(**final** Object lock1, **final** Object lock2) {

**return** **new** Thread() {

**public** **void** run() {

**synchronized** (lock1) {

System.***out***.println(Thread.*currentThread*() + " acquired a lock on " + lock1 + " and wants a lock on " + lock2);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

**throw** **new** RuntimeException(e);

}

**synchronized** (lock2) {

System.***out***.println(Thread.*currentThread*() + " acquired both locks.");

}

}

}

};

}

}

4. Explain why recursive implementation of QuickSort will require O(log n) of additional space

**private** **static** **void** quickSort (**int**[] array, **int** left, **int** right) {

**int** index = partition(array, left, right);

//Sort left half

**if** (left < index - 1)

quickSort(array, left, index - 1);

//Sort right half

**if** (index < right)

quickSort(array, index , right);

}

**private** **static** **int** partition (**int** array[], **int** left, **int** right) {

**int** pivot = array[(left + right) / 2]; //Pick pivot point

**while** (left <= right) {

//Find element on left that should be on right

**while** (array[left] < pivot)

left++;

//Find element on right that should be on left

**while** (array[right] > pivot)

right--;

//Swap elements and move left and right indices

**if** (left <= right) {

**int** temp = array[left];

array[left] = array[right];

array[right] = temp;

left++;

right--;

}

}

**return** left;

}

Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most O(log n) nested recursive calls, it uses O(log n) space. Any sorting algorithm that operates on a contiguous array requires O⁢(log ⁡n) extra space, since this is the number of bite [sic] required to represent an index into the array.

5. Explain the design pattern used in Java and .NET io stream/reader APIs.

The decorator pattern is used in Java and .NET io classes when you manipulated input/output streams (and the same applies for readers and writers). Inputstream, bytearrayinputstream, stringbuilderinputstreams and so on are based elements. Filterinputstream is the base class for the decorator classes. Filter input streams (such as bufferedinput stream) can do additional things when they read streams or write to them.

ex: new BufferedReader( new FileInputStream() ).readLine();

Decorator pattern has four components

1. ***Component:*** The Component defines the interface for objects that can have responsibilties added dynamically
2. ***ConcreteComponent:*** It is simply an implementation of Component interface
3. ***Decorator:*** The Decorator has a reference to a Component, and also conforms to theComponent interface. Decorator is essentially wrapping the Component
4. ***ConcreteDecorator:*** The ConcreteDecorator just adds responsibilities to the original Component.

Now let's map these concepts to java.io pacakge classes.

Component:

InputStream :

This abstract class is the superclass of all classes representing an input stream of bytes.

ConcreteComponent:

A FileInputStream obtains input bytes from a file in a file system. What files are available depends on the host environment.

FileInputStream is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

Decorator:

A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.

ConcreteDecorator:

A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods.

**Code Test**  
  
Create an Iterator filtering framework:

(1) IObjectTest interface with a single boolean test(Object o) method

|  |
| --- |
| Package iterators; |
|  |  |
|  | public interface IObjectTest { |
|  | boolean test(Object o); |
|  | } |

(2) An implementation of Iterator (let's call it FilteringIterator) which is initialized with another Iterator and an IObjectTest instance: new FilteringIterator(myIterator, myTest).

Your FilteringIterator will then allow iteration over 'myIterator', but skipping any objects which don't pass the 'myTest' test. Create a simple unit test for this framework.

**package** iterators;

**import** java.util.Iterator;

**public** **class** FilterIterator **implements** Iterator<Object> {

**private** **final** Iterator<Object> wrapped;

**private** **final** IObjectTest predicate;

**private** Object next;

FilterIterator(Iterator<Object> wrapped, IObjectTest predicate) {

**this**.wrapped = wrapped;

**this**.predicate = predicate;

}

@Override

**public** **boolean** hasNext() {

**while** (next == **null** && wrapped.hasNext()) {

next = wrapped.next();

**if** (predicate.test(next))

**return** **true**;

next = **null**;

}

**return** next != **null**;

}

@Override

**public** Object next() {

**if** (next == **null**)

hasNext();

**try** {

**return** next;

} **finally** {

next = **null**;

hasNext();

}

}

@Override

**public** **void** remove() {

wrapped.remove();

}

}