**Interface or Abstract Class**

**What is the difference between an Abstract Class and an Interface?**  
Well, isn't it an interesting question or for few, a foolish question?  
  
For a number of Java newbies, the only difference between two is, "You can't have method body in Interface.". Well, I would say that it is technically (Implementation vise) correct, but doesn't reflect the concept.  
  
**Abstract Class**  
An abstract class, is the most generalized form of an Entity hierarchy (In Java, a Class is an entity), such that it doesn't represent a real entity on its own, but provides a basis on which real entities can be build within that hierarchy. An abstract class provides a default behavior and also provide abstract behavior (Operations that must be override by the real classes).  
  
**Interface**  
An interface, on the other hand is a contract that states that you have to agree to provide following operations if you agree to abide by me, i.e. an entity that agrees to abide by the contract of interface, will have to provide the behavior (operations) declared in interface.  
  
**Interface or Abstract Class**  
Next question is when to use abstract class and when use an Interface? Though it many a depends on requirements as well, but I follow a thumb rule,

* If I have to build a class hierarchy, I should consider generalization. Many a time I will reach to most crude form of generalization, i.e. at an abstract level. I would declare abstract methods and will also define common behavior for few of the methods. This would be my abstract class.
* On the other hand, if I want to expose a particular behavior to the client, then I would define an interface.

Further, I would suggest that one should go through different Java APIs, and spend some time understanding when and why they used interface or abstract class. Java APIs are excellent source for learning on building design blocks for any OOPs language.

**Elastic Load Balancing with Sticky Sessions**

## What is Session Affinity (Sticky Sessions)? Why is it in High Demand?

When you only have one application server talking to your clients life is easy: all the session contexts can be stored in that application server’s memory for fast retrieval. But in the world of highly available and scalable applications there’s likely to be more than one application server fulfilling requests, behind a load balancer. The load balancer routes the first request to an application server, who stores the session context in its own memory and gives the client back a cookie. The next request from the same client will contain the cookie – and, if the same application server gets the request again, the application will rediscover the session context. But what happens if that client’s next request instead gets routed to a different application server? That application server will not have the session context in its memory – even though the request contains the cookie, the application can’t discover the context.

If you’re willing to modify your application you can overcome this problem. You can store the session context in a shared location, visible to all application servers: the database or memcached, for example. All application servers will then be able to lookup the cookie in the central, shared location and discover the context. Until now, this was the approach you needed to take in order to retain the session context behind an Elastic Load Balancer.

But not all applications can be modified in this way. And not all developers want to modify existing applications. Instead of modifying the application, you need the load balancer to route the same client to the same application server. Once the client’s request has been routed to the correct application server, that application server can lookup the session cookie in its own memory and recover the conversational context.

**That’s what sticky sessions are**: the load balancer routing the same client to the same application server. And that’s why they’re so important: If the load balancer supports sticky sessions then you don’t need to modify your application to remember client session context.

@Override public String toString() {

StringBuilder result = new StringBuilder();

String newLine = System.getProperty("line.separator");

result.append( this.getClass().getName() );

result.append( " Object {" );

result.append(newLine);

//determine fields declared in this class only (no fields of superclass)

Field[] fields = this.getClass().getDeclaredFields();

//print field names paired with their values

for ( Field field : fields ) {

result.append(" ");

try {

result.append( field.getName() );

result.append(": ");

//requires access to private field:

result.append( field.get(this) );

}

catch ( IllegalAccessException ex ) {

System.out.println(ex);

}

result.append(newLine);

}

result.append("}");

return result.toString();

}

<http://www.javapractices.com/topic/TopicAction.do?Id=55>

**Why we need a Sorting in Data Structure**

Sorting makes it possible to search a particular data element in a collection quickly by applying the binary search technique.

Everyone knows what is the binary search technique? We are using it every time when we look up a word in a dictionary.

How much time we can save by using the binary search technique? Let’s do a rough calculation. Assume that we have a sorted dictionary and an un-sorted dictionary with the same collection of words printed on about 1000 pages.

Looking up a word in the un-sorted dictionary, the best case is that you found the word on the first page; the worst case is that you looked through all 1000 the pages, and found the word on the last page. So roughly the average time you spend is checking 500 pages.

Looking up a word in the sorted dictionary, the best case is that you found the word on the center page, which is the first page you will be checking based on the binary search technique; the worst case is that you found the word on the final page after you have divided the dictionary 10 times. So roughly the average time you spend is checking 5 pages.

The answer is obvious. We are saving about 99% of our time by using the sorted dictionary!