**In java using math, check if a number is a palindrome**

public static boolean isPalindrome(int number) { // 12345

int palindrome = number; //copy number into variable

int reverse = 0;

while(palindrome != 0) {

int remainder = palindrome % 10; //remainder represents every digit, ‘5’, ‘4’, ‘3’, ‘2’, ‘1’

reverse = remainder + 10\*reverse; //‘5’, ’54’, ‘543’, ‘5432’, ‘54321’

palindrome = palindrome / 10; //‘1234’, ‘123’, ’12’, ‘1’, 0

}

if(number == reverse) {

return true;

}

return false;

}

**How do you reverse the words in a string? Code**

ex: “Hello World” => “olleH dlroW”

Method 1:

public String reverseString(String str) {

String finalStr = “”, temp = “”;

for(int i = 0; i <= str.length() - 1; i++) {

temp += str.charAt(i);

if(str.charAt(i) == ‘ ‘ || i == str.length() - 1) { //if we meet “ “ or end of the string

for(int j = temp.length() -2; j>=0; j—) {

finalStr += temp.charAt(j);

if((j == 0) && (i != str.length()-1)) //deal with “ “

finalStr += “ “;

}

temp = “”; //reset temp

} // end if

} // end for

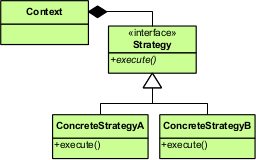
}

**Explain Strategy Pattern, Singleton pattern**

Strategy pattern is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern:

1. defines a family of algorithms,
2. encapsulates each algorithm, and
3. makes the algorithms interchangeable within that family

Strategy lets the algorithm vary independently from clients that use it.

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other factors. These factors are not known for each case until run-time.

**Explain way to optimize SQL**

1 Isolate long-running queries. 2 identify the cause of long-running queries.

**How does the garbage collector know if an object can be collected?**

Garbage Collector will define a set of roots and determine reachability from the roots. An object is reachable if there is some path of references from the roots by which the executing program can access the object. The root are always accessible to the program. Any object that are reachable from the roots are considered “live”. Objects that are not reachable are considered garbage cause they can no longer affect the future course of program execution.

**In java, How to handle out of memory error?**

Increase heap size or look up for any memory leak

**Fatest way to find the middle node in a linked list**

Note: should be clear what “middle node” means when there are even number nodes in the linked-list.

Use two pointers. Move one at twice speed of the second one. When the first pointer reaches the end of the list, the second one will be pointing to the middle.

Node \*FindMiddle(Node \*head) {

Node \*ptr1, \*ptr2;

ptr1 = ptr2 = head;

int i = 0;

while(ptr1->next != null) {

if(i == 0) {

ptr1 = ptr1->next;

i = 1;

if(i == 1) {

ptr2 = ptr2->next;

ptr1 = ptr1->next;

i = 0;

} // end if

} // end while

return ptr2;

}

**Find the first index of the substring.**

public int findIndex(String mainStr, String subString) {

if(mainStr == null || subString == null) return -1;

int mainLength = mainStr.length();

int subLength = subString.length();

if(mainLength < subLength) return -1;

for(int i=0; i < mainLength; i++) {

char curr = mainStr.charAt(i);

if(curr == subString.charAt(0)) {

if(i + subLength <= mainLength) {

if(mainStr.substring(i, subLength + i).equals(subString))

return i;

}

}

}

}

return -1;

}

**Design an online restaurant reservation system. Customers should be able to reserve the table for a timeslot (Sort of like OpenTable)**

**why it is said a good practice to override hashcode() of an object when you override equals()**

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashcode() method must consistently return the same integer. If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. Otherwise a violation of the general contract for Object.hashCode will occur.

1. 首先equals()和hashcode()这两个方法都是从object类中继承过来的。   
equals()方法在object类中定义如下：   
  public boolean equals(Object obj) {   
return (this == obj);   
}   
很明显是对两个对象的地址值进行的比较（即比较引用是否相同）。但是我们必需清楚，当String 、Math、还有Integer、Double。。。。等这些封装类在使用equals()方法时，已经覆盖了object类的equals（）方法。比如在String类中如下：   
  public boolean equals(Object anObject) {   
if (this == anObject) {   
    return true;   
}   
if (anObject instanceof String) {   
    String anotherString = (String)anObject;   
    int n = count;   
    if (n == anotherString.count) {   
char v1[] = value;   
char v2[] = anotherString.value;   
int i = offset;   
int j = anotherString.offset;   
while (n-- != 0) {   
    if (v1[i++] != v2[j++])   
return false;   
}   
return true;   
    }   
}   
return false;   
}   
很明显，这是进行的内容比较，而已经不再是地址的比较。依次类推Double、Integer、Math。。。。等等这些类都是重写了equals()方法的，从而进行的是内容的比较。当然了基本类型是进行值的比较，这个没有什么好说的。   
我们还应该注意，Java语言对equals()的要求如下，这些要求是必须遵循的：   
• 对称性：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”。   
• 反射性：x.equals(x)必须返回是“true”。   
• 类推性：如果x.equals(y)返回是“true”，而且y.equals(z)返回是“true”，那么z.equals(x)也应该返回是“true”。   
• 还有一致性：如果x.equals(y)返回是“true”，只要x和y内容一直不变，不管你重复x.equals(y)多少次，返回都是“true”。   
• 任何情况下，x.equals(null)，永远返回是“false”；x.equals(和x不同类型的对象)永远返回是“false”。   
以上这五点是重写equals()方法时，必须遵守的准则，如果违反会出现意想不到的结果，请大家一定要遵守。   
2. 其次是hashcode() 方法，在object类中定义如下：   
  public native int hashCode();   
说明是一个本地方法，它的实现是根据本地机器相关的。当然我们可以在自己写的类中覆盖hashcode()方法，比如String、Integer、Double。。。。等等这些类都是覆盖了hashcode()方法的。例如在String类中定义的hashcode()方法如下：   
    public int hashCode() {   
int h = hash;   
if (h == 0) {   
    int off = offset;   
    char val[] = value;   
    int len = count;   
  
            for (int i = 0; i < len; i++) {   
                h = 31\*h + val[off++];   
            }   
            hash = h;   
        }   
        return h;   
}   
解释一下这个程序（String的API中写到）：   
s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]   
使用 int 算法，这里 s[i] 是字符串的第 i 个字符，n 是字符串的长度，^ 表示求幂。（空字符串的哈希码为 0。）   
  
3.这里我们首先要明白一个问题：   
equals()相等的两个对象，hashcode()一定相等；   
equals（）不相等的两个对象，却并不能证明他们的hashcode()不相等。换句话说，equals()方法不相等的两个对象，hashcode()有可能相等。（我的理解是由于哈希码在生成的时候产生冲突造成的）。   
反过来：hashcode()不等，一定能推出equals()也不等；hashcode()相等，equals()可能相等，也可能不等。解释下第3点的使用范围，我的理解是在object、String等类中都能使用。在object类中，hashcode()方法是本地方法，返回的是对象的地址值，而object类中的equals()方法比较的也是两个对象的地址值，如果equals()相等，说明两个对象地址值也相等，当然hashcode()也就相等了；在String类中，equals()返回的是两个对象内容的比较，当两个对象内容相等时，   
Hashcode()方法根据String类的重写（第2点里面已经分析了）代码的分析，也可知道hashcode()返回结果也会相等。以此类推，可以知道Integer、Double等封装类中经过重写的equals()和hashcode()方法也同样适合于这个原则。当然没有经过重写的类，在继承了object类的equals()和hashcode()方法后，也会遵守这个原则。   
  
4.谈到hashcode()和equals()就不能不说到hashset,hashmap,hashtable中的使用，具体是怎样呢，请看如下分析：   
Hashset是继承Set接口，Set接口又实现Collection接口，这是层次关系。那么hashset是根据什么原理来存取对象的呢？   
在hashset中不允许出现重复对象，元素的位置也是不确定的。在hashset中又是怎样判定元素是否重复的呢？这就是问题的关键所在，经过一下午的查询求证终于获得了一点启示，和大家分享一下，在java的集合中，判断两个对象是否相等的规则是：   
1)，判断两个对象的hashCode是否相等   
      如果不相等，认为两个对象也不相等，完毕   
      如果相等，转入2)   
（这一点只是为了提高存储效率而要求的，其实理论上没有也可以，但如果没有，实际使用时效率会大大降低，所以我们这里将其做为必需的。后面会重点讲到这个问题。）   
2)，判断两个对象用equals运算是否相等   
      如果不相等，认为两个对象也不相等   
      如果相等，认为两个对象相等（equals()是判断两个对象是否相等的关键）   
为什么是两条准则，难道用第一条不行吗？不行，因为前面已经说了，hashcode()相等时，equals()方法也可能不等，所以必须用第2条准则进行限制，才能保证加入的为非重复元素。   
比如下面的代码：   
  
public static void main(String args[]){   
String s1=new String("zhaoxudong");   
String s2=new String("zhaoxudong");   
System.out.println(s1==s2);//false   
System.out.println(s1.equals(s2));//true   
System.out.println(s1.hashCode());//s1.hashcode()等于s2.hashcode()   
System.out.println(s2.hashCode());   
Set hashset=new HashSet();   
hashset.add(s1);   
hashset.add(s2);   
/\*实质上在添加s1,s2时，运用上面说到的两点准则，可以知道hashset认为s1和s2是相等的，是在添加重复元素，所以让s2覆盖了s1;\*/   
Iterator it=hashset.iterator();   
            while(it.hasNext())   
            {   
             System.out.println(it.next());   
            }   
最后在while循环的时候只打印出了一个”zhaoxudong”。   
输出结果为：false   
            true   
            -967303459   
            -967303459   
这是因为String类已经重写了equals()方法和hashcode()方法，所以在根据上面的第1.2条原则判定时，hashset认为它们是相等的对象，进行了重复添加。   
但是看下面的程序：   
import java.util.\*;   
public class HashSetTest   
{   
   public static void main(String[] args)   
    {   
                 HashSet hs=new HashSet();   
                 hs.add(new Student(1,"zhangsan"));   
                 hs.add(new Student(2,"lisi"));   
                 hs.add(new Student(3,"wangwu"));   
                 hs.add(new Student(1,"zhangsan"));   
    
                 Iterator it=hs.iterator();   
                 while(it.hasNext())   
                 {   
                        System.out.println(it.next());   
                 }   
     }   
}   
class Student   
   {   
     int num;   
     String name;   
     Student(int num,String name)   
                {   
                this.num=num;   
                 this.name=name;   
                 }   
              public String toString()   
                {   
                    return num+":"+name;   
                 }   
           }        
输出结果为：   
                      1:zhangsan   
                   1:zhangsan   
                   3:wangwu   
                   2:lisi   
问题出现了，为什么hashset添加了相等的元素呢，这是不是和hashset的原则违背了呢？回答是：没有   
因为在根据hashcode()对两次建立的new Student(1,"zhangsan")对象进行比较时，生成的是不同的哈希码值，所以hashset把他当作不同的对象对待了，当然此时的equals()方法返回的值也不等（这个不用解释了吧）。那么为什么会生成不同的哈希码值呢？上面我们在比较s1和s2的时候不是生成了同样的哈希码吗？原因就在于我们自己写的Student类并没有重新自己的hashcode()和equals()方法，所以在比较时，是继承的object类中的hashcode()方法，呵呵，各位还记得object类中的hashcode()方法比较的是什么吧！！   
它是一个本地方法，比较的是对象的地址（引用地址），使用new方法创建对象，两次生成的当然是不同的对象了（这个大家都能理解吧。。。），造成的结果就是两个对象的hashcode()返回的值不一样。所以根据第一个准则，hashset会把它们当作不同的对象对待，自然也用不着第二个准则进行判定了。那么怎么解决这个问题呢？？   
答案是：在Student类中重新hashcode()和equals()方法。   
例如：   
  class Student   
{   
int num;   
String name;   
Student(int num,String name)   
{   
            this.num=num;   
            this.name=name;   
}   
public int hashCode()   
{   
            return num\*name.hashCode();   
}   
public boolean equals(Object o)   
{   
            Student s=(Student)o;   
            return num==s.num && name.equals(s.name);   
}   
public String toString()   
{   
            return num+":"+name;   
}   
}   
根据重写的方法，即便两次调用了new Student(1,"zhangsan")，我们在获得对象的哈希码时，根据重写的方法hashcode()，获得的哈希码肯定是一样的（这一点应该没有疑问吧）。   
当然根据equals()方法我们也可判断是相同的。所以在向hashset集合中添加时把它们当作重复元素看待了。所以运行修改后的程序时，我们会发现运行结果是：   
                      1:zhangsan   
                   3:wangwu   
                   2:lisi   
可以看到重复元素的问题已经消除。   
关于在hibernate的pojo类中，重新equals()和hashcode()的问题：   
1)，重点是equals，重写hashCode只是技术要求（为了提高效率）   
2)，为什么要重写equals呢，因为在java的集合框架中，是通过equals来判断两个对象是否相等的   
3)，在hibernate中，经常使用set集合来保存相关对象，而set集合是不允许重复的。我们再来谈谈前面提到在向hashset集合中添加元素时,怎样判断对象是否相同的准则，前面说了两条，其实只要重写equals()这一条也可以。   
但当hashset中元素比较多时，或者是重写的equals()方法比较复杂时，我们只用equals()方法进行比较判断，效率也会非常低，所以引入了hashcode()这个方法，只是为了提高效率，但是我觉得这是非常有必要的（所以我们在前面以两条准则来进行hashset的元素是否重复的判断）。   
比如可以这样写：   
public int hashCode(){   
   return  1;}//等价于hashcode无效   
这样做的效果就是在比较哈希码的时候不能进行判断，因为每个对象返回的哈希码都是1，每次都必须要经过比较equals()方法后才能进行判断是否重复，这当然会引起效率的大大降低。   
我有一个问题，如果像前面提到的在hashset中判断元素是否重复的必要方法是equals()方法（根据网上找到的观点），但是这里并没有涉及到关于哈希表的问题，可是这个集合却叫hashset，这是为什么？？   
我想，在hashmap,hashtable中的存储操作，依然遵守上面的准则。所以这里不再多说。这些是今天看书，网上查询资料，自己总结出来的，部分代码和语言是引述，但是千真万确是自己总结出来的。有错误之处和不详细不清楚的地方还请大家指出，我也是初学者，所以难免会有错误的地方，希望大家共同讨论

**Simulate a deadlock**

In order for a deadlock to occur, you must have the following four conditions met:

1. Mutual Exclusion: Only one process can use a resource at a given time.
2. Hold and Wait: A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. No preemption: One process cannot forcibly remove another process’ resource.
4. Circular wait: two or more processes form a circular chain where each process is waiting on another resource in the chain.

create a deadlock in sql server:

CREATE TABLE deadlock\_example\_table\_1 (column1 int)

CREATE TABLE deadlock\_example\_table\_2 (column2 int)

INSERT INTO deadlock\_example\_table\_1 (column1)

SELECT 1

UNION ALL

SELECT 2

UNION ALL

SELECT 3

GO

INSERT INTO deadlock\_example\_table\_2 (column2)

SELECT 1

UNION ALL

SELECT 2

UNION ALL

SELECT 3

GO

Open two query windows in SQL Server Management Studio

Execute this in Query Window 1:

BEGIN TRAN

DELETE FROM deadlock\_example\_table\_1 WHERE column = 2

Execute this in Query Window 2:

BEGIN TRAN

DELETE FROM deadlock\_example\_table\_2 WHERE column = 2

DELETE FROM deadlock\_example\_table\_1 WHERE column = 2

Execute this in Query Window 1:

DELETE FROM deadlock\_example\_table\_2 WHERE column = 2

The transaction in Query window 1 has been chosen as the deadlock victim, and will be rolled back.

COMMIT or ROLLBACK transaction in Query Window 2

**Given 2 unsorted integer arrays, get the intersection of the 2.**

read oracle doc for HashSet. Main method: add(E e), contains(object o), remove(object o)

We pick one of the array and load it into hash implemented data structure, HashSet and then proceeds further to find intersection of elements.

Since hashed data structure’s complexity is O(1), the total complexity to find intersection of elements the complexity would become

Time complexity: O(m) + O(n)\*O(1)

Space Complexity: O(m)

public void intersect1(int[] a, int[] b) {

HashSet<Integer> hs = new HashSet<Integer>();

for(int i = 0; i < b.length; i++) {

hs.add(b[i]);

}

for(int i = 0; i < a.length; i++) {

if(hs.contains(a[i])) System.out.println(a[i]);

}

pros: best algorithm when compared to all others provided one implements appropriate hashcode method

cons: when the size of the data structure grows too high, it might lead to hash collisions.

When two arrays are sorted:

For every element in array A, do a binary search in array B, so here for every value in ‘A’ we go through log(n) iterations in ‘B’ to find out if element in ‘A’ exists in’B’

Time complexity: O(m)\*O(logn)

see oracle doc for binarySearch(int[] a, int key)

public void intersect(int[] a, int[] b) {

for(int i=0; i < a.length; i++) {

int val = binarySearch(b, a[i]);

if(val >=0 ) {

System.out.println(b[val]);

break;

}

}

}

**"How to find a special weight ball from 8 balls while other 7 have the same weight with a balance? (all same color, shape etc.) Trick is that you don't know if the special ball is lighter or heavier.”**

Weigh(123 and 456) //Try-1

Case1

if(123 == 456)

weigh(7 and 8) //Try-2

Case2

Weigh(123 and 456)

if(123 > 456)

weigh(1 and 2) //Try-2

if(1 == 2) 3 is heavy

Case3

if(123 < 456)

weigh(4 and 5) //Try-2

if(4 == 5) 6 is heavy

In two tries you can find out the heaviest ball.

**How to reverse a linked-list**

Reverse a linked list iterative algorithm

1. The head node’s next pointer should be set to NULL since the head will become the tail. This is an exception for the head node, and can be done outside the while loop. But, before we do this we will need a temp variable to point to the 2nd node (the node after the head node), because the only way to reference the 2nd node is through the head node’s next pointer.

2. The 2nd node (the node after the head node) should have it’s own next pointer changed to point to the head node. This will reverse the order of the nodes. But, remember that the 2nd node’s next pointer will at first be pointing to the 3rd node. This means that before we change the 2nd node’s next pointer, we have to save a reference to the 3rd node otherwise we will have no way of referencing the 3rd node. So, we simply store a reference to the 3rd node in a variable before we change the 2nd node’s next pointer.

3. The 3rd node then becomes the “first” node in the while loop and we repeat the process of changing pointers described in step 2.

4. Continue step 3 until we come across a node that has a next pointer set to NULL. When we do come across a NULL next pointer we just set the head node to point to the node that has the NULL next pointer. This node was previously the tail node, but is now the head node because we are reversing the linked list.

Iteration:

public void reverse(Node head) {

if(head == null || head->next == null)

return; //empty or just one node in list

Node second = head.next;

Node third = second.next;

seond.next = head; //second now points to head

head.next = null; //change head pointer to null

if(third == null) //if only two nodes, we are done

return;

Node current = third;

Node previous = second;

while(current != null) { //or start from the third node

Node temp = current.next;

current.next = previous;

if(temp == null) head = current; //if we arrive at the tail, set head to last node

previous = current; //move previous and current to next node

current = temp;

}

}

recursion:

Reverse a linked list recursive Java

Now, let’s find a recursive solution in Java for this problem. The advantage that we have with recursion is the fact that we can go to the very end of the linked list and work backwards to reverse the list starting from the every end. This is because with recursion we can go to the very end of the linked list, but also essentially “save” our place in each and every node we traverse so that we can go backwards and change the pointers in the linked list so that the list is reversed – if you are rusty on recursion you can read our Recursion tutorial.

Base case for recursive solution

Before we come up with our recursive case, we should come up with our base case for this problem. The base case is what we use to essentially stop the recursion from continuously running. So, when should we stop the recursion from running – try to see if you can come up with this scenario on your own.

The base case for this problem will actually occur whenever we reach the tail node of the linked list – and by “tail” node we mean the very last node. We will know that we have reached the tail node when the current node’s next node is NULL – so the current node will be the tail node.

What should be done in the base case for this recursive problem?

Now, we know what the base case is and how to check for it, but what exactly should we be doing in this case? Well, we obviously want to have a return statement so that we can put a stop to the recursion. But, is there anything else that we should be doing here as well? It turns out that there is something else we need to do in the base case, because in the base case we are at the very end of the linked list. This means that because we are trying to reverse the list, we need to set the head pointer to point to the very last node.

This means that our base case would look like this – remember that we are assuming that in the recursive case, we will just be passing in the very next node in the linked list to the recursive function:

Java code for recursive solution’s base case:

/\* if we are at the TAIL node:

\*/

if(currentNode.next == NULL)

{

//set HEAD to TAIL since we are reversing list

head = currentNode;

return; //since this is the base case

}

Coming up with a recursive case

Now that we’ve come up with the base case in Java, what about the recursive case? Well, in the recursive case we will clearly need to change the pointers so that the nodes are reversed. For a given node – let’s say we are dealing with node 99 in the linked list image above – the Node coming after node 99 (node 37) is represented by Node 99 -> next. If we want node 37 to point back to node 99 (which is what we would want if we are reversing the nodes), then we would set the next pointer of node 37 to point back to Node 99, which in pseudocode would look like Node99 -> next -> next = Node 99.

We would also need to get rid of the pointer from node 99 to node 37 so we would have to set Node 99 -> next to NULL.

So, now we can come up with this code that would be our final recursive answer to the problem of reversing a singly linked list:

public void recursiveReverse(Node startnode) {

if(startnode == null) return;

if(startnode->next = null) { //set head to tail since we are reversing

head = startnode;

return;

}

recursiveReverse(startnode->next);

startnode->next->next = startnode; //change the pointer direction

startnode->next = null; //set old pointer to null

}

The cool thing about recursion in this problem is that it essentially allows us to iterate through the linked list backwards – even though this is impossible in a singly linked list just by using the pointers (or course this would certainly be possible in a doubly linked list). The reason we can do this in recursion is because of the fact that the stack frames for each function call are saved until we get to the very end of the list, and then it is as if we are ‘unwinding’ the stack frames one by one going backwards in the linked list – where each stack frame essentially represents a node in the linked list. Make sure you understand this point, because it is important and really helps in your understanding of how to use recursion to solve problems.

**Difference between array and link list**

1. First and major difference between linked list and array data structure is that former doesn't support random access, while later support random access. linked list is sequential, in order to retrieve an element, you need to traverse till that, while if you know index, you can retrieve an element from array very quickly, because it doesn't involved traversal.

2. Second major difference between array and linked-list data structure is that, array needs contiguous memory allocation, which may result in java.lang.OutOfMemoryError: Java Heap Space if there is not enough contiguous ( a big chunk) of memory in Java Heap. On the other hand, linked list is distributed data structure, it's element are scattered over heap and doesn't need a contiguous memory allocation. This makes linked list ideal, if you have scattered memory.

3. Third major difference is fixed length, array is a fixed length data structure, you provide length or size of array at the time of creation, later you can not modify that size. On the other hand, linked list is dynamic data structure, it can grow and doesn't required size to be specified at the time of creation, because each node keep tracks of other.

4. It's easy to insert and delete elements from linked list than array, especially inserting element at beginning of linked list, and deleting element from end of linked list is O(1) operation. On the other hand array is fixed length data structure, so memory is allocated during initialization, and doesn't really change due to addition and removal of elements. Though you can set a particular index null, to cut the reference count of that object.

5. Array is ideal for implementing fast caches e.g. HashMap or Hashtable, which requires constant time retrieval e.g. Map data structure provides O(1) performance for get(Key key) operation, while linked list based structure provides liner performance i.e. O(n) for retrieval operation, where n is the number of elements in linked list.

6. Array can be one or multi-dimensional, while linked list can be singly, doubly or circular linked list. Two dimensional array are most common in multi-dimensional and used to represent matrix in Java. You can use two dimensional array to represent a plain of x,y coordinates, frequently used in Game programming. Java programming language provides support for creating array at syntax level, it supports both single and multidimensional array. Java API also provides a class called java.util.LinkedList, which is an implementation of doubly linked list data structure.

That's all on my list of differences between array and linked list data structure. I strongly suggest to get a good hold of these data structure, especially linked list, which is very popular among data structure interview questions. Questions like appending elements into linked list, deleting elements, reversing linked list are quite common in various programming jobs. At very least, knowledge of fundamental data structure is essential to do well in programming jobs.

Read more: http://javarevisited.blogspot.com/2013/07/difference-between-array-and-linked-list-java.html#ixzz2z72WasWS

test a vending machine.

**The difference between arraylist and linkedlist**

linkedList vs ArrayList in Java

Difference between LinkedList and ArrayList in JavaAll the differences between LinkedList and ArrayList has there root on difference between Array and LinkedList data-structure. If you are familiar with Array and LinkedList data structure you will most likely derive following differences between them:

1) Since Array is an index based data-structure searching or getting element from Array with index is pretty fast. Array provides O(1) performance for get(index) method but remove is costly in ArrayList as you need to rearrange all elements. On the Other hand LinkedList doesn't provide Random or index based access and you need to iterate over linked list to retrieve any element which is of order O(n).

2) Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array

and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java. ArrayList also needs to update its index if you insert something anywhere except at the end of array.

3) Removal is like insertions better in LinkedList than ArrayList.

4) LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

**What is Polymorphism?**

it is the ability to process objects of various types and classes through a single, uniform interface.

Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding(. Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.

For instance, let’s consider a class Animal and let Cat be a subclass of Animal. So any cat IS animal. Here, Cat satisfies the IS-A relationship for its own type as well as its super class Animal.

Note: It’s also legal to say every object in Java is polymorphism in nature, as each one passes an IS-A test for itself and also for object class.

Static Polymorphism:

In Java, static polymorphism is achieved through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.

At compile time, Java knows which method to invoke by checking the method signatures. So, this is called compile time polymorphism or static binding. The concept will be clear from the following example:

class DemoOverload {

public int add(int x, int y) {

return x + y;

}

public int add(int x, int y, int z) {

return x + y + z;

}

public int add(double x, int y) {

return (int)x + y;

}

public int add(int x, double y) {

return x + (int)y;

}

}

class Test {

public static void main(String [] args) {

DemoOverload demo = new DemoOverload();

demo.add(2, 3);

demo.add(2,3,4);

demo.add(2,3,4);

demo.add(2,5,3);

}

}

Dynamic Polymorphism:

Suppose a sub class overrides a particular method of the super class. Let’s say, in the program we create an object of the subclass and assign it to the super class reference. Now, if we call the overridden method on the super class reference then the sub class version of the method will be called.

class Vehicle {

public void move() {

System.out.println(“Vehicles can move!”);

}

}

class MotorBike extends Vehicle {

public void move() {

System.out.println(“MotorBike can move!”);

}

}

class Test {

public static void main(String[] args) {

Vehicle vh = new MotorBike();

vh.move();

vh = new Vehicle();

vh.move();

}

}

**Synchronize**

**How to implement a thread? Give me a common example of thread in java api. How could a thread wake up from a sleeping thread? How to test a feature with multiple thread?**

Thread is normally a piece of codes used to deal with concurrence. Single thread cannot solve problems when multiple events or actions need to occur at the same time. For example, a program is not capable of drawing pictures while reading keystrokes.

Two way of creating a thread

1 extend from the Thread class

Import java.lang.\*;

Public class Counter extends Thread {

Public void run() {

…

}

}

run() method is where all the work of the Counter class thread is done.

2 implement Runnable interface

Import java.lang.\*;

Public class Counter implements Runnable {

Thread T;

public void run() {

…

}

}

Start thread:

ThreadTest1 tt = new ThreadTest1();

tt.start();

ThreadTest2 tt = new ThreadTest2();

Thread t = new Thread(tt);

t.start();

State:

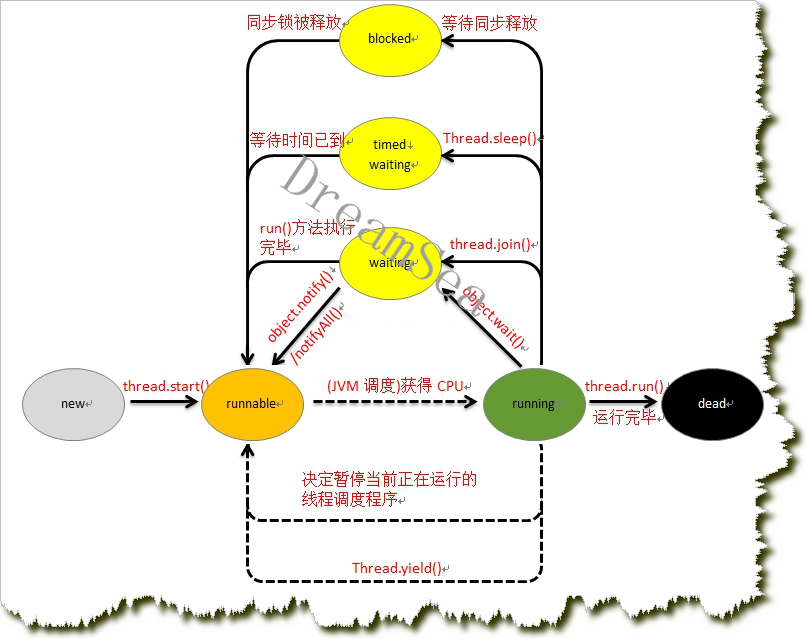
新生状态（New）： 当一个线程的实例被创建即使用new关键字和Thread类或其子类创建一个线程对象后，此时该线程处于新生(new)状态，处于新生状态的线程有自己的内存空间，但该线程并没有运行，此时线程还不是活着的（not alive）；

就绪状态（Runnable）： 通过调用线程实例的start()方法来启动线程使线程进入就绪状态(runnable)；处于就绪状态的线程已经具备了运行条件，但还没有被分配到CPU即不一定会被立即执行，此时处于线程就绪队列，等待系统为其分配CPCU，等待状态并不是执行状态； 此时线程是活着的（alive）；

运行状态（Running）： 一旦获取CPU(被JVM选中)，线程就进入运行(running)状态，线程的run()方法才开始被执行；在运行状态的线程执行自己的run()方法中的操作，直到调用其他的方法而终止、或者等待某种资源而阻塞、或者完成任务而死亡；如果在给定的时间片内没有执行结束，就会被系统给换下来回到线程的等待状态；此时线程是活着的（alive）；

阻塞状态（Blocked）：通过调用join()、sleep()、wait()或者资源被暂用使线程处于阻塞(blocked)状态；处于Blocking状态的线程仍然是活着的（alive）

死亡状态（Dead）：当一个线程的run()方法运行完毕或被中断或被异常退出，该线程到达死亡(dead)状态。此时可能仍然存在一个该Thread的实例对象，当该Thready已经不可能在被作为一个可被独立执行的线程对待了，线程的独立的call stack已经被dissolved。一旦某一线程进入Dead状态，他就再也不能进入一个独立线程的生命周期了。对于一个处于Dead状态的线程调用start()方法，会出现一个运行期(runtime exception)的异常；处于Dead状态的线程不是活着的（not alive）。



线程的方法（Method）、属性（Property）

1）优先级（priority）

每个类都有自己的优先级，一般property用1-10的整数表示，默认优先级是5，优先级最高是10；优先级高的线程并不一定比优先级低的线程执行的机会高，只是执行的机率高；默认一个线程的优先级和创建他的线程优先级相同；

2）Thread.sleep()/sleep(long millis)

当前线程睡眠/millis的时间（millis指定睡眠时间是其最小的不执行时间，因为sleep(millis)休眠到达后，无法保证会被JVM立即调度）；sleep()是一个静态方法(static method) ，所以他不会停止其他的线程也处于休眠状态；线程sleep()时不会失去拥有的对象锁。 作用：保持对象锁，让出CPU，调用目的是不让当前线程独自霸占该进程所获取的CPU资源，以留一定的时间给其他线程执行的机会；

3）Thread.yield()

  让出CPU的使用权，给其他线程执行机会、让同等优先权的线程运行（但并不保证当前线程会被JVM再次调度、使该线程重新进入Running状态），如果没有同等优先权的线程，那么yield()方法将不会起作用。

4）thread.join()

 使用该方法的线程会在此之间执行完毕后再往下继续执行。

5）object.wait()

  当一个线程执行到wait()方法时，他就进入到一个和该对象相关的等待池(Waiting Pool)中，同时失去了对象的机锁—暂时的，wait后还要返还对象锁。当前线程必须拥有当前对象的锁，如果当前线程不是此锁的拥有者，会抛出IllegalMonitorStateException异常,所以wait()必须在synchronized block中调用。

6）object.notify()/notifyAll()

  唤醒在当前对象等待池中等待的第一个线程/所有线程。notify()/notifyAll()也必须拥有相同对象锁，否则也会抛出IllegalMonitorStateException异常。

7）Synchronizing Block

 Synchronized Block/方法控制对类成员变量的访问；Java中的每一个对象都有唯一的一个内置的锁，每个Synchronized Block/方法只有持有调用该方法被锁定对象的锁才可以访问，否则所属线程阻塞；机锁具有独占性、一旦被一个Thread持有，其他的Thread就不能再拥有（不能访问其他同步方法），方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。

**What is a tree? What is a balance tree? Traverse a tree.**

**How to remove HTML tags from a string? What is the test case of this feature? If you can only test one case, what is it? if you can test two cases, what is the second one?**

**Describe about your classes you've taken**

**Figure out test cases for a blackbox testing for a file-uploading system**

d

**clockwise print out a M\*N matrix. Solution: recursive**

**Balance binary tree**

d

**Sorting algorithm**

**what difference between abstract class and interface?**

First, an interface can only contain abstract methods and /or static final variables(constants). Abstract class, on the other hand, can implement methods and contain variables that are not constants.

Second, an interface cannot implement any methods. A class that implements an interface must implement all methods defined in that interface. An interface can extend from other interfaces, and can extend from multiple interfaces.

The idea behind this is that interface is a contract that force any class to implement a set of behaviors or functions while abstract class is used to be inherited by other class.

**Arrays and vectors?**

Vector is resizeable but Array is not.

Difference between Arraylist and Vector

1 Synchronization: Vectors are synchronized. Any method that touches the Vector’s content is thread safe. ArrayList, on the other hand, is unsynchronized, making them, therefore, not thread safe. With that difference in mind, using synchronization will incur a performance hit.

2 Data growth: when you insert an element into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its size by 50 percent. If you know the rate at which it grows, Vector does possess a slight advantage since you can set the increment value.

**String and String buffer?**

String class is used to manipulate character strings that cannot be changed. Simply stated, objects of type String are read only and immutable.

StringBuffer is a thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

The principal operations on a StringBuffer are the append and insert methods, which are overloaded so as to accept data of any type.

**Inheritance**

A class that is derived from another class is called a subclass. The class from which the subclass is derived is called a superclass.

Excepting Object, which has no superclass, every class has one and only one direct superclass.

When you want to create a new class and there is already a class that includes some of the code you want, you can derive your new class from the existing class. In this way, you can reuse the fields and methods of the existing class.

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructor are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from subclass.

**Advantages of OO?**

OOP provides a clear modular structure for programs.

OOP makes it easy to maintain and modify existing code as new objects can be created with small difference to existing ones.

OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer.

Then he gave me two things to test?

1. string

2. vending machine.

write a function that would detect if a sorted array contains two integer that sum up to 7. And then improve your code so that the array is accessed with only one iteration