**How to swap two variables without using third var?**

|  |  |  |
| --- | --- | --- |
| a=a+b; b=a-b; a=a-b; | a = a^b b= a^b a= a^b | a=a\*b; b=a/b; a=a/b; |

**How does the Jsp Expressions works? What happens at the back?**

The code placed within **JSP expression tag** is written to the output stream of the response. So you need not write out.print() to write data. It is mainly used to print the values of variable or method.

Jsp expressions is a normal java expression, when we write the Jsp expression in our Jsp page like  
<%=some expression %> then in the servlet i.e. generated by Jsp compiler the Jsp expression will be placed without " “ in out.print() method. For example :  
  
<%int i=10,j=20,k=30;%>k----> <%=k%>  
  
in the servlet equivalent to above Jsp page   
  
out.print("k---->");  
out.print(k);  
  
Like that at the back the Jsp expression will be placed in the out.print() without " " so that the value of the expression will be evaluated and printed.

Difference between Enum and Iterator?

|  |  |
| --- | --- |
| **Enumeration** | **Iterator** |
| Using *Enumeration*, you can only traverse the collection. You can’t do any modifications to collection while traversing it. | Using *Iterator*, you can remove an element of the collection while traversing it. |
| *Enumeration* is introduced in JDK 1.0 | *Iterator* is introduced from JDK 1.2 |
| *Enumeration* is used to traverse the legacy classes like *Vector*, *Stack* and *HashTable*. | *Iterator* is used to iterate most of the classes in the collection framework like *ArrayList*, *HashSet*, *HashMap*, *LinkedList* etc. |
| Methods : *hasMoreElements()*and *nextElement()* | Methods : *hasNext()*, *next()* and *remove()* |
| *Enumeration* is fail-safe in nature. | *Iterator* is fail-fast in nature. |
| *Enumeration* is not safe and secured due to it’s fail-safe nature. | *Iterator*is safer and secured than *Enumeration*. |

**How many web.config an application can have?**

You can create as many as web.config files in an application restricted to one web.config per folder, any folder cannot have more than web.config.

Also the web.config at root folder is differ from other web.config files exist is other folder. The web.config at root basically contains all necessary detail for application functioning. While other web.config inside other folder rather than root folder must not contains these details.

**Write a program on palindrome?**

|  |  |
| --- | --- |
| 1. **class** PalindromeExample{ 2. **public** **static** **void** main(String args[]){ 3. **int** r,sum=0,temp; 4. **int** n=454;//Input 5. temp=n; 6. **while**(n>0){ 7. r=n%10;  //getting remainder 8. sum=(sum\*10)+r; 9. n=n/10; 10. } 11. **if**(temp==sum) 12. System.out.println("palindrome"); 13. **else** 14. System.out.println("not palindrome"); 15. } 16. } | boolean isPalindrome(String str) {    int n = str.length();    for( int i = 0; i < n/2; i++ )  if (str.charAt(i) != str.charAt(n-i-1))  return false;  return true;  }  **Also**  public static boolean istPalindrom(char[] word){  int i1 = 0;  int i2 = word.length - 1;  while (i2 > i1) {  if (word[i1] != word[i2]) {  return false;  }  ++i1;  --i2;  }  return true;  } |

Write a program to generate Fibonacci series?

|  |
| --- |
| 1. **import** java.util.Scanner; 2. **public** **class** Fibonacci 3. { 4. **public** **static** **void** main(String[] args) 5. { 6. **int** n, a = 0, b = 0, c = 1; 7. Scanner s = **new** Scanner(System.in); 8. System.out.print("Enter value of n:"); 9. n = s.nextInt(); 10. System.out.print("Fibonacci Series:"); 11. **for**(**int** i = 1; i <= n; i++) 12. { 13. a = b; 14. b = c; 15. c = a + b; 16. System.out.print(a+" "); 17. } 18. } 19. } |
| Enter value of n:5  Fibonacci Series:0 1 1 2 3 |

**Singleton in Java?**

|  |
| --- |
| public enum Foo {  INSTANCE;  }  Which internally will be treated like  public class Foo {  // It will be our sole hero  private static final Foo INSTANCE = new Foo();  } |
| public enum Singleton {  SINGLE;  public void myMethod(){  }  }  and then just have your threads use your instance like:  Singleton.SINGLE.myMethod(); |

|  |
| --- |
| public class MySingleton {  private static class Loader {  static final MySingleton INSTANCE = new MySingleton();  }  private MySingleton () {}  public static MySingleton getInstance() {  return Loader.INSTANCE;  }  }  It works because it uses the **class loader** to do all the synchronization for you for free: The class MySingleton.Loader is first accessed inside the getInstance() method, so the Loader class loads when getInstance() is called for the first time.  Further, the class loader guarantees that all static initialization is complete before you get access to the class - that's what gives you thread-safety. |

What happens when we do like this? i.e., what will be the output? 

|  |  |
| --- | --- |
| String s = "Hello";   s.concat("world");   System.out.println(s); | It prints Hello, concatenation doesn’t happen. |

If we modify the question like this what happends? 

|  |  |
| --- | --- |
| String s = null;   s.concat(“ABC”);   System.out.println(s.toLowerCase()); | It throws Null pointer exception, since no operation can be performed with null value. |

How many ways you can iterate collections? What are they?

4 ways:

1. **Iterable.forEach method (Java 8)**

Recently introduced in java 8, this method can be called on any Iterable and takes one argument implementing the functional interface java.util.function.Consumer. e.g.

Collection<String> collection = Arrays.asList("How", "To", "Iterate", "In", "Java");

collection.forEach(s -> System.out.println(s));

1. **Java “foreach” loop (Java 5)**

The “foreach” loop syntax is:

for (Type var : Iterable<Type>) {

// do something with "var"

}

e.g.

|  |
| --- |
| Collection<String> collection = Arrays.asList("How", "To", "Iterate", "In", "Java");    for(String s : collection) {      System.out.println(s);  }   1. **java.util.Iterator (Java 2)** |
| Using Iterator is still very popular way to iterate over collections, mainly due to addtional methods it provide to manipulate the collection elements.  Collection<String> collection = Arrays.asList("How", "To", "Iterate", "In", "Java");    Iterator<String> itr = collection.iterator();    while(itr.hasNext()) {      System.out.println(itr.next());  } |
|  |
| 1. **Traditional for loop**   This is most appropriate when dealing with indexed collections such as list. It uses the standard  for loop invented in the early 1970s in the C language.  List<String> list = Arrays.asList("How", "To", "Iterate", "In", "Java");    for( int i=0; i < list.size(); i++ )  {      System.out.println(list.get(i));  } |
| **Which method will be called when we call the add() at 1 & at 2**  public class A   {   public void add(int a, int b){......}   public void add(float a, float b){.....}   public void add(double a, double b){.....}     public static void main(String ar[]){   A a = new A();   a.add(1,2);----------à1   a.add(1.0,2.0); -------à2   }   } |
|  |

1 will invoke int method, 2 will invoke double method

**How remove() will work in HashSet class?**

Under the covers, HashSet uses HashMap, which calls HashMap.removeEntryForKey(Object) when either HashSet.remove(Object) or Iterator.remove() is called. This method uses both hashCode() and equals() to validate that it is removing the proper object from the collection.

**What is the difference between Hashtable and HashMap?**

|  |  |
| --- | --- |
| **HashTable** | **HashMap** |
| Hashtable is synchronized, slower, more memory | Not Synchronized, faster, less memory |
| Hashtable does not allow null keys or values. | HashMap allows one null key and any number of null values. |
| HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object. | Hashmap object values are iterated by using iterator |
| Legacy Class, Extends AbstractMap class | Not a legacy class, Extends Dictionary class |
| Preferred in multi-threaded applications, now not preferred because ConcurrentHashMap is a better option | HashMap is preferred in single threaded applications. If you want to use it in multi threaded wrap it using Collections.SynchronizedMap() method. |

How to reverse a list?

Collections.reverse(yourList);

|  |
| --- |
| /\* Function to reverse the linked list \*/  static void reverse(struct Node\*\* head\_ref)  {      struct Node\* prev   = NULL;      struct Node\* current = \*head\_ref;      struct Node\* next;      while (current != NULL)      {          next  = current->next;          current->next = prev;          prev = current;          current = next;      }      \*head\_ref = prev;  } |

**What are invariant variables?**

Invariant means something that should stick to its conditions no matter whatever changes or whoever uses/transforms it. That is to say, a property of a class always fulfills or satisfies some condition even after going through transformations by using public methods. So, the client or user of this class is ensured about the class and its property.

For example,

1. condition on function argument is that, it should always be > 0 (greater than zero) or should not be null.
2. minimum\_account\_balance property of an account class states, it cannot go below 100. So all public functions should respect this condition and ensure class invariant.
3. rule based dependency between variables, that is, value of one variable depends on another, so if one changes, using some fix-rule, other must also change. This relationship between 2 variables must be preserved. If it does not, then invariant is violated.

Class invariants are methods which check the validity of an object's state (its data). The idea is to define validation methods for fields, and to perform these validations whenever the fields change. As usual, this should be done without repeating any code.

**Java supports pass by value or pass by reference?**

Java is always **pass-by-value**. Unfortunately, they decided to call the location of an object a "reference". When we pass the value of an object, we are passing the reference to it. This is confusing.

|  |
| --- |
| public static void main( String[] args ) {  Dog aDog = new Dog("Max");  // we pass the object to foo  foo(aDog);  // aDog variable is still pointing to the "Max" dog when foo(...) returns  aDog.getName().equals("Max"); // true, java passes by value  aDog.getName().equals("Fifi"); // false  }  public static void foo(Dog d) {  d.getName().equals("Max"); // true  // change d inside of foo() to point to a new Dog instance "Fifi"  d = new Dog("Fifi");  d.getName().equals("Fifi"); // true  } |
| the Dog "Fifi" as the object reference is passed by value. If it were passed by reference, then the aDog.getName() in main would return "Fifi" after the call to foo.  in Java everything is passed by value. Primitives are passed by value, and object references are passed by value. The objects themselves are never passed to a method, but the objects are always in the heap and only a reference to the object is passed to the method. |

|  |
| --- |
| public static void main( String[] args ) {  Dog aDog = new Dog("Max");  foo(aDog);  // when foo(...) returns, the name of the dog has been changed to "Fifi"  aDog.getName().equals("Fifi"); // true  }  public static void foo(Dog d) {  d.getName().equals("Max"); // true  // this changes the name of d to be "Fifi"  d.setName("Fifi");  } |
| In the above example, FiFi is the dog's name after call to foo(aDog) because the object's name was set inside of foo(...). Any operations that foo performs on d are such that, for all practical purposes, they are performed on aDog itself (except when d is changed to point to a different Doginstance like d = new Dog("Boxer")). |

**What is difference between GenericServlet and HttpServlet?**

The GenericServlet is protocol independent whereas HttpServlet is HTTP protocol specific. HttpServlet provides additional functionalities such as state management etc.

**Difference between forward() method and sendRedirect() method ?**

|  |  |
| --- | --- |
| **forward() method** | **sendRedirect() method** |
| 1) forward() sends the same request to another resource. | 1) sendRedirect() method sends new request always because it uses the URL bar of the browser. |
| 2) forward() method works at server side. | 2) sendRedirect() method works at client side. |
| 3) forward() method works within the server only. | 3) sendRedirect() method works within and outside the server. |

**RequestDispatcher in Servlet**

The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration.

There are two methods defined in the RequestDispatcher interface.

1. Forward - Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. Include - Includes the content of a resource (servlet, JSP page, or HTML file) in the response.

|  |
| --- |
| 1. RequestDispatcher rd=request.getRequestDispatcher("servlet2"); 2. rd.forward(request, response); |
| 1. RequestDispatcher rd=request.getRequestDispatcher("/index.html"); 2. rd.include(request, response); |