FLOW CONTROL AND EXCEPTION HANDLING
www.techfaq360.com
• Write code using if and switch statements and identify legal argument types for these statements.

Consider the following if-else block:

```
public void method1(boolean flag) //0
{
  boolean bool = false;
 int i = 20;
 if(flag)
 {
  System.out.println("first if");
 }
 else if(flag == bool)
 {
  System.out.println("first else-if");
 }
 else if( bool )
 {
  System.out.println("second else if");
 }
 else
 {
  System.out.println("last else");
 }

 if( i = 30 ){ ... } //invalid because the expr. i = 30 returns a an int value 30
 if( i == 30){ ...  } //valid
 if( bool = true ){ ... } //valid because the expr. bool = true returns a boolean value true
 int k;  //Note that k is uninitialized here.
 bool = true;
 if(bool)
 {
   k = 100; //Note that k is being initialized here. And this will always get executed as bool is true;
 }
 System.out.println(k); //Still this will NOT work.
}
```

Rules:
1. An if condition MUST be a boolean. If you put an expression in the if condition, the value of that expression must be of type boolean.
2. In a chained if : else-if : else construct, the 'else' attaches to the nearest 'if' above it. So, the

above construct is equiv. to:

```
3.
4.  if(flag)
5.  {
6.   System.out.println("first if");
7.  }
8.  else
9.  {
10.   if(flag == bool)
11.   {
12.    System.out.println("first else-if");
13.   }
14.   else
15.   {
16.    if( bool )
17.    {
18.     System.out.println("second else if");
19.    }
20.    else
21.    {
22.     System.out.println("last else");
23.    }
24.   }
25.  }
26.
```

Consider the following switch block:

```
public void method1(integraltype i) //0
{
  int k = 30;
 switch(i)
 {
  case 100 :  System.out.println(100); //1
  case 150 :  System.out.println(150); break;  //2 invalid if i is of type bytecase 3.2 :
System.out.println(i); //3 invalid
  default  :  System.out.println(i); //position of default may be anywhere.
  case 'c' :  System.out.println(i); //4
  case true : System.out.println(i); //5 invalid
  case k   :  System.out.println(i); //6 invalid
```

```
 }
}
```

Rules:
1. 'i' could be of type byte, char, short, int. It cannot be of type long, float, double, boolean or any kind of Object. So, //3 and //5 are invalid.
2. The case value should be a compile time constant. So, //6 is not valid.
3. The case value should 'fit' into the type of the case variable. Ex. if 'i' is of type byte then //2 is invalid.
4. 'break' after every case is not necessary. If it is not there after a case, then the control falls through until a break or end of case. So, if i is 100, the above block will print 100 and 150.

• Write code using all forms of loops including labeled and unlabeled use of break and continue, and state the values taken by loop control variables during and after loop execution.

Write code that makes proper use of exceptions and exception handling clauses (try, catch, finally) and declares methods and overriding methods that throw exceptions.

Consider the following method:

```
public void method1() throws IOException //0
{
 try
 {
  throw new FileNotFoundException("FNE");
  //System.out.println("This line Will never get executed"); //Compile time error : Unreachable code.
 }
 catch(NullPointerException ne)
 {
  throw new FileNotFoundException("FNE");
 }
 catch(FileNotFoundException fe)
 {
   //This block will NOT catch FileNotFoundException thrown by the catch block above.
   //This block will ONLY catch FileNotFoundException thrown by the try block.
   throw new EOFException("EOF");
 }
 finally
 {
   //This block has nothing to do with exception catching. That means if you throw an exception in try or catch block, either you have to provide an appropriate catch block or declare it in the throw clause of the method.
   //It provides a way to the programmer to make sure that code in this block always executes.
   //Executes even if there is a return in try/catch blocks.
   //DOES NOT EXECUTE if there is System.exit() in try/catch blocks.
 }
```

}

Concept: There are lot of rules regarding what exceptions can a method throw and they all seem to be very confusing. But they all stem from the same concept, Plug and Play. New code should not break pre-existing code.
Consider a component (not an AWT component but any s/w component) C that has a method m. Method m has declared IOException in it's throws clause. Say there is another component U, that uses C (and calls m). It needs to catch IOException, which it does and every thing is fine. Now, instead of C, a subclass of C (say NewC) is supplied to U. As NewC is a subclass of C, U shouldn't feel a difference. It's ok even if the method m of NewC, starts throwing FileNotFoundException because FileNotFoundEx is a subclass of IOException and U is already catching it. It is somethig like this:

```
//in a user component U
{
  ...
  C c = getC(); //somehow gets the component C. It may actually  be  NewC.
  try
  {
    c.m(); //can potentially throw IOException. So needs a try/catch block.
  }
  catch(IOException ioe) //It'll work even if m() throws FileNotFoundException.
  {
    //handle exception
  }
  ...
}
```

The above code in U will fail if m() throws some other exception like Exception or say, java.rmi.RemoteException. Now, it should be pretty clear to you that the restrictions imposed on the overriding method about exception are to promote component based architecture which allow you to treat s/w as components which you can take out and replace.
In short, an overriding method can throw any subclass of exceptions declared in the the throws clause of the superclass's method. It CANNOT throw super class exception or any new exception.
Can an overriding method have no throws clause if the overridden method does have one? Imagine such a situation and view it from the perspective shown above and you should get your answer. Hint: Think about whether the above code will fail or not if m() of NewC throws no exception.