Tokenizer Clean-up

The class Tokenizer needs to be clean in the following points:

1. The types of tokens need to be centralized. They can be easily to be modified because they are packed into one location. Since, the token types are used in the future as the precedence of operators, they are numbered accordingly. We add a new class of the constants:

```java
class Precedence {
    final int UNDEF = -1;
    final int LEFTPAREN = 1;
    final int RIGHTPAREN = 2;
    final int NUMBER = 3;
    final int VARIABLE = 4;
    final int PLUS = 5;
    final int MINUS = 6;
    final int MULT = 7;
    final int DIV = 8;
}
```

2. All precedence values are replaced in the method GetTokens(). Moreover, the default constructor TokenAndType with type = -1 must not be removed because the value -1 can not replaced by UNDEF that is a non-static value. Therefore in the future, the calling statement TokenAndType() can be used to initialized the default case with tokenName = null and tokenType = -1.

3. The program has an error when the expression is invalid. It will be an infinite loop because the switch statement did not cover the default case properly. It was default: break;
   It should be corrected as follows:
   ```java
default: System.out.print("invalid expression");System.exit(0);
   ```
   If we want to inform the user the location of the error within the expression, we can even modify further.
   ```java
default:
    System.out.println("invalid expression, only following tokens are found.");
    PrintTokens();
    System.exit(0);
   ```

4. The method GetTokens(0 did not cover the case of the hidden multiplication operator (*) when the number is followed by variable or x or vice-versa. To handle these cases (123 x or x 123), we need to add a new variable currentType that holds the type of the found token. It is declared and initialized as follows:
   ```java
   int currentType = p.UNDEF;
   ```
   When a token is discovered, it is saved to a linked list the token and its type; and then currentType is re-set to the type of new discovered token. Before this currentType is reset, it holds the type of the previous token. This information is very valuable for us to find down the hidden operator within the expression. The hidden operators are possible in the following 2 cases:
   ```java
   - currentType is NUMBER and the new token type is VARIABLE
   - currentType is VARIABLE and the new token type is NUMBER
   ```
   If one of these cases happens, the hidden operator * with type MULT is saved in the linked list, then the new token is saved after, and finally currentType is reset.
Here is a complete GetTokens() method:

```java
// method in tokenizerLP.java
public void GetTokens()
{

Precedence p = new Precedence(); // get the constants of precedence

int currentType = p.UNDEF;

NodePtr temp = null;
NodePtr last = null; // points to the last node
String token=""; // from string of characters, cut into 1 token
// at a time

InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader console = new BufferedReader(reader);
System.out.print("please enter data> ");
String StrExp = null;
try
{
    StrExp = console.readLine();
}
catch (IOException e)
{
    System.out.println("we call an" + e);
    System.exit(1);
}

char firstChar=' ';  
if (StrExp.length()!=0)firstChar = StrExp.charAt(0);

while(StrExp!=null)
{

switch (firstChar)
{
    case '.': case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7': case '8': case '9':
        do {
            token=token+firstChar;
            if (StrExp.length()==1){StrExp = null;}
            else
            {
                StrExp=StrExp.substring(1);
                firstChar = StrExp.charAt(0);
            }
        }while( ((firstChar >='0' && firstChar <= '9')
            || firstChar == '.')
```
&& StrExp != null);

    // try to find the hidden token *
    if (currentType == p.VARIABLE)
      { //create a new node for the hidden token
        temp = new NodePtr("*", p.MULT);
        // link to top and last
        if (last == null) // no node
          { top = temp;
            last = temp; }
        else { last.next = temp;
                  last = temp; }
      }

    //create a new node for a new token
    temp = new NodePtr(token, p.NUMBER);
    currentType = p.NUMBER;
    // link to top and last
    if (last == null) // no node
      { top = temp;
        last = temp; }
    else { last.next = temp;
            last = temp; }

    token = "; // to reset
    break;

    case ": // blanks
    while(firstChar==")")
      {
        if (StrExp.length()==1){StrExp = null;}
        else
          {
            StrExp=StrExp.substring(1);
            firstChar = StrExp.charAt(0);
          }
      }
    break;

    case ": case ": case ": case ": case ": case ": case "/
    case 'x':
    if (firstChar == 'x')
      { // try to find the hidden token *
        if (currentType == p.NUMBER)
          { //create a new node for the hidden token
            temp = new NodePtr("*", p.MULT);
            // link to top and last
            if (last == null) // no node
              { top = temp;
                last = temp; }
            else { last.next = temp;
                      last = temp; }
      }
else { last.next = temp;  
        last = temp;  
    }
    
    // end of hidden

    temp = new NodePtr("x", p.VARIABLE);  
    currentType = p.VARIABLE;  
}

else if (firstChar == '(')
    { temp = new NodePtr("(", p.LEFTPAREN);  
        currentType = p.LEFTPAREN;  
    }
else if (firstChar == ')')
    { temp = new NodePtr(")", p.RIGHTPAREN);  
        currentType = p.RIGHTPAREN;  
    }
else if (firstChar == '+')
    { temp = new NodePtr("+", p.PLUS);  
        currentType = p.PLUS;  
    }
else if (firstChar == '-')
    { temp = new NodePtr("-", p.MINUS);  
        currentType = p.MINUS;  
    }
else if (firstChar == '*')
    { temp = new NodePtr("*", p.MULT);  
        currentType = p.MULT;  
    }
else { temp = new NodePtr("/", p.DIV);  
        currentType = p.DIV;  
    }

    // insert here

    if (last == null) // no node
        { top = temp;  
            last = temp;  
        }
else { last.next = temp;  
        last = temp;  
    }
    
    // end of insert

    if (StrExp.length()==1){StrExp = null;}
else
    {
        StrExp=StrExp.substring(1);
        firstChar = StrExp.charAt(0);
    }
    break;

default:
    System.out.println("invalid expression, only following tokens are found:");
    printTokens();
    System.exit(0);

    } // end of switch statement

    } // end of the bigger loop

} // end of GetTokens