Converting **postfix to infix** involves reversing the process of infix-to-postfix conversion. Here's how it works:

Algorithm for Postfix to Infix Conversion

1. **Initialize a Stack**:
   - Use a stack to store intermediate expressions.
2. **Scan the Postfix Expression**:
   - Read the expression from left to right.
   - For each symbol:
     - **Operand**: Push it onto the stack.
     - **Operator**: Pop the top two elements from the stack, combine them with the operator in between, and push the resulting expression back onto the stack.
3. **Final Expression**:
   - After processing all symbols, the stack will contain a single element: the resulting infix expression.

Example 1

Postfix Expression:
`A B C * + D E F ^ - /`

Steps:

1. Read `A`: Push onto stack → `Stack = [A]`
2. Read `B`: Push onto stack → `Stack = [A, B]`
3. Read `C`: Push onto stack → `Stack = [A, B, C]`
4. Read `*`: Pop `C` and `B`, form `(B * C)`, push back → `Stack = [A, (B * C)]`
5. Read `+`: Pop `(B * C)` and `A`, form `(A + (B * C))`, push back → `Stack = [(A + (B * C))]`
6. Read `D`: Push onto stack → `Stack = [(A + (B * C)), D]`
7. Read `E`: Push onto stack → `Stack = [(A + (B * C)), D, E]`
8. Read `F`: Push onto stack → `Stack = [(A + (B * C)), D, E, F]`
9. Read `^`: Pop `F` and `E`, form `(E ^ F)`, push back → `Stack = [(A + (B * C)), D, (E ^ F)]`
10. Read `-`: Pop `(E ^ F)` and `D`, form `(D - (E ^ F))`, push back → `Stack = [(A + (B * C)), (D - (E ^ F))]`
11. Read `/`: Pop `(D - (E ^ F))` and `(A + (B * C))`, form `((A + (B * C)) / (D - (E ^ F)))`, push back → `Stack = [((A + (B * C)) / (D - (E ^ F)))]`

Infix Expression:

```
((A + (B * C)) / (D - (E ^ F)))
```

Example 2

Postfix Expression:
`A B + C D + *`

Steps:

1. Read `A`: Push onto stack → `Stack = [A]`
2. Read `B`: Push onto stack → `Stack = [A, B]`
3. Read `+`: Pop `B` and `A`, form `(A + B)`, push back → `Stack = [(A + B)]`
4. Read `C`: Push onto stack → `Stack = [(A + B), C]`
5. Read `D`: Push onto stack → `Stack = [(A + B), C, D]`
6. Read `+`: Pop `D` and `C`, form `(C + D)`, push back → `Stack = [(A + B), (C + D)]`
7. Read `*`: Pop `(C + D)` and `(A + B)`, form `((A + B) * (C + D))`, push back → `Stack = [((A + B) * (C + D))]`

Infix Expression:

```
((A + B) * (C + D))
```

Example 3

Postfix Expression:
`A B C ^ + D *`

Infix Expression:

```
((A + (B ^ C)) * D)
```

Example 4

Postfix Expression:
`A B + C D * -`

Infix Expression:

```
((A + B) - (C * D))
```

Example 5

Postfix Expression:
`A B C + D * E - /`

Infix Expression:

```
(A / (((B + C) * D) - E))
```

Java Code for Postfix to Infix Conversion

```
import java.util.Stack;

public class PostfixToInfix {
    public static String postfixToInfix(String postfix) {
        Stack<String> stack = new Stack<>();

        for (char ch : postfix.toCharArray()) {
            // Operand: Push onto stack
            if (Character.isLetterOrDigit(ch)) {
                stack.push(String.valueOf(ch));
            }
            // Operator: Pop two elements, form "(operand1 operator operand2)"
            else {
                String operand2 = stack.pop();
                String operand1 = stack.pop();
                String expression = "(" + operand1 + " " + ch + " " + operand2 +
")";
                stack.push(expression);
            }
        }

        // The final expression on the stack is the result
        return stack.pop();
    }

    public static void main(String[] args) {
        String postfix = "A B C * + D E F ^ - /";
        System.out.println("Infix: " + postfixToInfix(postfix));
    }
}
```

Output:

```
Infix: ((A + (B * C)) / (D - (E ^ F)))
```