Objective:

This document describes the new Java formatter algorithm implemented to fix the eclipse bug [https://bugs.eclipse.org/bugs/show_bug.cgi?id=303519](https://bugs.eclipse.org/bugs/show_bug.cgi?id=303519). The major issue we tried to fix was line limit not being followed by old formatter as explained in the bug.

How New Algorithm works?

CodeFormatterVisitor is implementation of ASTVisitor which will parse through code tree. Depending on the type of node in the AST, appropriate visit(node) function is called.

The old algorithm would handle over-limit lines by throwing an AlignmentException, and then attempting to re-process the node. This made it difficult to track what formatting had been attempted.

The new algorithm stores the state information and uses it to format the code. So in the new algorithm the Scribe does not directly print, but rather records the node which would be printed and the alignment state of that node. The formatting state of the code statement (or statement fragment) is stored in the Scribe.printState (which will be referred as ‘print state’ in this document). The printState is an array of BasicState.

The important types of the State are:

1. PrintState: Used to store information about the content of code
2. SpaceState: Stores information about whitespace
3. IndentState: Stores indentation information
4. AlignmentState: Stores the information about alignment (explain with example)

In essence, CodeFormatterVisitor calls various print methods Scribe.print() which earlier used to output the code required, but now simply saves the state in the ‘print state’ as a pending write.

After we get to end of recording (either with the end of code block or newline), the pending writes are processed and output in Scribe.processPrintState(). This function processes the ‘print state’ we have stored in the Scribe.

First, Scribe.checkLineWrap() function finds out whether we need any line breaks due to exceeding line limits (if max line lengths are enabled). If the line limit is being exceeded, we iterate through the pending writes, trying different combinations of line breaks until we find the best one. Scribe.tryBreakAlignmentState() records the attempts which we have made.

Any modifications to the code being formatted are done using inserting/removing
OptimizedReplaceEdits.

We haven't changed much in how comment formatting works except for a few minor fixes. So for comment formatting in this new algorithm- the 'print state' is "paused" when we encounter comment. We output everything that's in the state queue up to the comment, and then switch over to the "classic" comment formatting (using the restoring print state). When the comment is done, we go back to keeping state with the new way.

**Illustration:**
Consider an example:
`foo(arg1, bar(arg2));`

While formatting above statement, Scribe would contain following 'print state' at the end:
1. Enter Alignment state - Statement
2. Align Fragment with index 0
3. Print state - 'foo'
4. Print state - '('
5. Enter Alignment state - MessageArguments|Statement
6. Print state - 'arg1'
7. Align fragment
8. Enter Alignment state : MessageSend | MessageArguments | Statement
9. Align Fragment
10. Print state ','
11. Print 'bar'
12. Enter Alignment state - MessageArguments | MessageSend | MessageArgs | Statement
13. Align Fragment
14. Print '('
15. Print 'arg2'
16. Exit alignment state
17. Print ')'
18. Exit alignment state
19. Print ';
20. Exit alignment state
21. Restoring print state

`Scribe.printState()` will process this 'print state' after the end of statement.

**Terminology:**

1. **Alignment:**
   It keeps basic information about the alignment like location, indentation level, information about
breaks. Stack of Alignment objects is used to represent the Alignment State where the top of stack indicates the innermost alignment.

2. Fragment
Fragments are the pieces within an alignment. e.g. Parameters in the method calls are fragments. In the above statement `foo (arg1, bar(arg2))` the fragments in the outermost method call alignment are `arg1` and `bar(arg2)`.

3. Block break
Block break is when we try to break an alignment before its first fragment.
For example if we need to break:

```javascript
foo(arg1, bar (arg2));
```

And the line break occurs after `foo`

```javascript
foo(
  arg1, bar (arg2));
```

Then this will be referred to as a block break.

4. Tentative manager:
The tentative manager keeps track of the line breaks attempted by the formatter. It stores it as map of strings of attempted breaks.

So, for example, if we have this:

```javascript
doSomething(a, b, somethingElse(c, d));
```

And we attempt to format it as such:

```javascript
doSomething(
  a, b, somethingElse(c, d));
```

Here we would add a set of our attempted newlines, (e.g. `{ first element of doSomething method call }`) to the tentative map - if this fails (i.e. line still too long), we start over.

Next time we try to do the same line break, though, we first check the tentative map to see if that set of linebreaks (i.e. `{ first element of doSomething method call }`) has already been attempted. Because it has, we don't do a line break here, and instead try elsewhere.

**New Preferences for alignment**

We have also added new dynamic formatting options based on the number of elements.
The new format `FragmentDependentAlignment` for these preferences is tuple of three integers.

E.g. Consider the preference `alignment_for_parameters_in_method_declaration` which decides alignment preference for parameters in method declaration. Let's say the value of this pref is 16|4|48 which is basically `COMPACT_SPLIT|4|ONE_PER_LINE_SPLIT`.

So if the number of parameters in method declaration, they would be aligned in compact way else they would be split into one per line.

These preferences are backwards compatible. If the value of the above preference is just single integer in the configuration, it is used as the alignment preference and custom formatting based on number of elements would not be applicable.