# ripgrep crates/printer/src/json.rs: Code Companion

Reference code for the JSON Output lecture. Sections correspond to the lecture document.

---

## Section 1: The Configuration and Builder Pattern

```rust
/// The configuration for the JSON printer.
///
/// This is manipulated by the JSONBuilder and then referenced by the actual
/// implementation. Once a printer is build, the configuration is frozen and
/// cannot changed.
#[derive(Debug, Clone)]
struct Config {
    pretty: bool,                          // Emit pretty-printed JSON (multi-line)
    always_begin_end: bool,                // Emit begin/end even with no matches
    replacement: Arc<Option<Vec<u8>>>,     // Arc for cheap cloning, Option for "no
replacement"
}

impl Default for Config {
    fn default() -> Config {
        Config {
            pretty: false,
            always_begin_end: false,
            replacement: Arc::new(None),
        }
    }
}

/// A builder for a JSON lines printer.
#[derive(Clone, Debug)]
pub struct JSONBuilder {
    config: Config,
}

impl JSONBuilder {
    /// Return a new builder for configuring the JSON printer.
    pub fn new() -> JSONBuilder {
        JSONBuilder { config: Config::default() }
    }

    /// Create a JSON printer that writes results to the given writer.
    pub fn build<W: io::Write>(&self, wtr: W) -> JSON<W> {
        JSON {
            config: self.config.clone(),   // Clone severs builder-printer
relationship
            wtr: CounterWriter::new(wtr),
            matches: vec![],
        }
    }

    /// Print JSON in a pretty printed format.
    pub fn pretty(&mut self, yes: bool) -> &mut JSONBuilder {
        self.config.pretty = yes;
        self  // Return self for method chaining
    }

    /// When enabled, the `begin` and `end` messages are always emitted.
    pub fn always_begin_end(&mut self, yes: bool) -> &mut JSONBuilder {
        self.config.always_begin_end = yes;
```

```rust
        self    }    /// Set the bytes that will be used to replace each
  occurrence of a match.    pub fn replacement(         &mut self,
 replacement: Option<Vec<u8>>,     ) -> &mut JSONBuilder {
 self.config.replacement = Arc::new(replacement);         self    }}
```

The `Arc<Option<Vec<u8>>>` pattern allows distinguishing "no replacement configured" ( `None` ) from "replace with empty string" ( `Some(vec![])` ), while the `Arc` enables cheap cloning when building printers.

## Section 2: The JSON Printer Structure

```rust
#[derive(Clone, Debug)]
pub struct JSON<W> {
    config: Config,              // Frozen configuration from builder
    wtr: CounterWriter<W>,       // Tracks bytes written for statistics
    matches: Vec<Match>,         // Reusable buffer - cleared between uses
}

impl<W: io::Write> JSON<W> {
    /// Return a JSON lines printer with a default configuration.
    pub fn new(wtr: W) -> JSON<W> {
        // Delegate to builder - builder is the canonical construction path
        JSONBuilder::new().build(wtr)
    }
}

impl<W> JSON<W> {
    /// Returns true if and only if this printer has written at least one byte.
    pub fn has_written(&self) -> bool {
        self.wtr.total_count() > 0
    }

    /// Return a mutable reference to the underlying writer.
    pub fn get_mut(&mut self) -> &mut W {
        self.wtr.get_mut()
    }

    /// Consume this printer and return back ownership of the underlying writer.
    pub fn into_inner(self) -> W {
        self.wtr.into_inner()
    }
}
```

The `matches` vector is allocated once and reused across all searches, amortizing allocation costs. The `CounterWriter` wrapper tracks output volume for statistics reporting.

Section 3: The Sink Abstraction

```rust
/// An implementation of `Sink` associated with a matcher and an optional file
path.
///
/// Lifetime parameters:
/// * `'p` - lifetime of the file path (or `'static` if no path)
/// * `'s` - lifetime of the borrowed JSON printer
#[derive(Debug)]
pub struct JSONSink<'p, 's, M: Matcher, W> {
    matcher: M,                             // For finding match positions within
lines
    replacer: Replacer<M>,                  // For optional text substitution
    json: &'s mut JSON<W>,                  // Back-reference to parent printer
    path: Option<&'p Path>,                 // Optional file path for this search
    start_time: Instant,                    // For elapsed time calculation
    match_count: u64,                       // Matches found in current search
    binary_byte_offset: Option<u64>,        // Where binary data was detected
    begin_printed: bool,                    // Track if begin message was emitted
    stats: Stats,                           // Accumulated statistics
}

impl<W: io::Write> JSON<W> {
    /// Return an implementation of `Sink` for the JSON printer.
    /// No file path - will never print path with matches.
    pub fn sink<'s, M: Matcher>(
        &'s mut self,
        matcher: M,
    ) -> JSONSink<'static, 's, M, W> {
        JSONSink {
            matcher,
            replacer: Replacer::new(),
            json: self,
            path: None,   // No path association
            start_time: Instant::now(),
            match_count: 0,
            binary_byte_offset: None,
            begin_printed: false,
            stats: Stats::new(),
        }
    }

    /// Return an implementation of `Sink` associated with a file path.
    pub fn sink_with_path<'p, 's, M, P>(
        &'s mut self,
        matcher: M,
        path: &'p P,
    ) -> JSONSink<'p, 's, M, W>
    where
        M: Matcher,
        P: ?Sized + AsRef<Path>,   // Accept Path, PathBuf, str, etc.
    {
        JSONSink {
            matcher,
            replacer: Replacer::new(),
            json: self,
```

```
            path: Some(path.as_ref()),   // Convert to &Path
    start_time: Instant::now(),              match_count: 0,
    binary_byte_offset: None,                begin_printed: false,              stats:
    Stats::new(),           }     }}
```

The `?Sized` bound on `P` allows accepting both `Path` (unsized) and `PathBuf` (sized) through the `AsRef<Path>` trait. The sink borrows the printer mutably, ensuring exclusive access during a search.

## Section 4: The Wire Format and Message Types

```rust
impl<W: io::Write> JSON<W> {
    /// Write the given message followed by a new line.
    fn write_message(
        &mut self,
        message: &jsont::Message<'_>,
    ) -> io::Result<()> {
        // Choose compact or pretty output based on configuration
        if self.config.pretty {
            json::to_writer_pretty(&mut self.wtr, message)?;
        } else {
            json::to_writer(&mut self.wtr, message)?;
        }
        // Maintain JSON Lines format invariant: one object per line
        let _ = self.wtr.write(b"\n")?;
        Ok(())
    }
}

impl<'p, 's, M: Matcher, W: io::Write> JSONSink<'p, 's, M, W> {
    /// Write the "begin" message (idempotent - only writes once).
    fn write_begin_message(&mut self) -> io::Result<()> {
        if self.begin_printed {
            return Ok(());  // Already printed, skip
        }
        let msg = jsont::Message::Begin(jsont::Begin { path: self.path });
        self.json.write_message(&msg)?;
        self.begin_printed = true;
        Ok(())
    }
}

// From the Sink trait implementation - the begin callback
fn begin(&mut self, _searcher: &Searcher) -> Result<bool, io::Error> {
    self.json.wtr.reset_count();  // Reset byte counter for this search
    self.start_time = Instant::now();
    self.match_count = 0;
    self.binary_byte_offset = None;

    // Only emit begin immediately if always_begin_end is configured
    if !self.json.config.always_begin_end {
        return Ok(true);
    }
    self.write_begin_message()?;
    Ok(true)
}
```

The message types (`Begin`, `End`, `Match`, `Context`) are defined in the `jsont` module and implement `Serialize` for JSON output. The `begin_printed` flag prevents duplicate begin messages.

## Section 5: Handling Non-UTF-8 Data

From the extensive documentation in the source file:

```
/// ## Text encoding
///
/// JSON may only be encoded in UTF-8, UTF-16 or UTF-32. For the purposes of this
/// printer, we need only worry about UTF-8. The problem here is that searching
/// is not limited to UTF-8 exclusively, which in turn implies that matches
/// may be reported that contain invalid UTF-8.
///
/// Therefore, this printer will emit valid UTF-8 encoded bytes as normal
/// JSON strings and otherwise base64 encode data that isn't valid UTF-8. To
/// communicate whether this process occurs or not, strings are keyed by the
/// name `text` where as arbitrary bytes are keyed by `bytes`.
///
/// For example, when a path is included in a message, it is formatted like so,
/// if and only if the path is valid UTF-8:
///
/// ```json
/// {
///     "path": {
///         "text": "/home/ubuntu/lib.rs"
///     }
/// }
/// ```
///
/// If instead our path was `/home/ubuntu/lib\xFF.rs`, where the `\xFF` byte
/// makes it invalid UTF-8, the path would instead be encoded like so:
///
/// ```json
/// {
///     "path": {
///         "bytes": "L2hvbWUvdWJ1bnR1L2xpYv8ucnM="
///     }
/// }
/// ```
///
/// The printer guarantees that the `text` field is used whenever the
/// underlying bytes are valid UTF-8.
```

The dual representation (`text` vs `bytes`) ensures lossless data transfer: consumers can always recover original bytes either directly from the string or via base64 decoding.

---

## Section 6: Recording Matches Within Lines

```rust
impl<'p, 's, M: Matcher, W: io::Write> JSONSink<'p, 's, M, W> {
    /// Execute the matcher over the given bytes and record the match
    /// locations if the current configuration demands match granularity.
    fn record_matches(
        &mut self,
        searcher: &Searcher,
        bytes: &[u8],
        range: std::ops::Range<usize>,
    ) -> io::Result<()> {
        self.json.matches.clear();  // Reuse the buffer

        let matches = &mut self.json.matches;
        // find_iter_at_in_context handles context-aware match finding
        find_iter_at_in_context(
            searcher,
            &self.matcher,
            bytes,
            range.clone(),
            |m| {
                // Adjust offsets to be relative to line start, not buffer start
                let (s, e) = (m.start() - range.start, m.end() - range.start);
                matches.push(Match::new(s, e));
                true  // Continue finding matches
            },
        )?;

        // Don't report empty matches appearing at the end of the bytes
        // (prevents confusing output from patterns like `.*`)
        if !matches.is_empty()
            && matches.last().unwrap().is_empty()
            && matches.last().unwrap().start() >= bytes.len()
        {
            matches.pop().unwrap();
        }
        Ok(())
    }
}
```

The offset adjustment (`m.start() - range.start`) normalizes match positions to be relative to the line being reported, not the entire search buffer.

## Section 7: The Match and Context Sink Callbacks

```rust
impl<'p, 's, M: Matcher, W: io::Write> Sink for JSONSink<'p, 's, M, W> {
    type Error = io::Error;

    fn matched(
        &mut self,
        searcher: &Searcher,
        mat: &SinkMatch<'_>,
    ) -> Result<bool, io::Error> {
        self.match_count += 1;
        self.write_begin_message()?;  // Ensure begin was printed

        // Record individual match positions within this line
        self.record_matches(
            searcher,
            mat.buffer(),
            mat.bytes_range_in_buffer(),
        )?;
        // Compute replacement text if configured
        self.replace(searcher, mat.buffer(), mat.bytes_range_in_buffer())?;

        // Update statistics
        self.stats.add_matches(self.json.matches.len() as u64);
        self.stats.add_matched_lines(mat.lines().count() as u64);

        // Build submatches using small-vector optimization
        let submatches = SubMatches::new(
            mat.bytes(),
            &self.json.matches,
            self.replacer.replacement(),
        );

        let msg = jsont::Message::Match(jsont::Match {
            path: self.path,
            lines: mat.bytes(),
            line_number: mat.line_number(),
            absolute_offset: mat.absolute_byte_offset(),
            submatches: submatches.as_slice(),
        });
        self.json.write_message(&msg)?;
        Ok(true)  // Continue searching
    }

    fn context(
        &mut self,
        searcher: &Searcher,
        ctx: &SinkContext<'_>,
    ) -> Result<bool, io::Error> {
        self.write_begin_message()?;
        self.json.matches.clear();

        // Context lines may contain matches when search is inverted
        let submatches = if searcher.invert_match() {
            self.record_matches(searcher, ctx.bytes(), 0..ctx.bytes().len())?;
            self.replace(searcher, ctx.bytes(), 0..ctx.bytes().len())?;
```

```
            SubMatches::new(                    ctx.bytes(),
    &self.json.matches,                self.replacer.replacement(),                    )
    } else {            SubMatches::empty()          };            let msg =
    jsont::Message::Context(jsont::Context {                path: self.path,
    lines: ctx.bytes(),              line_number: ctx.line_number(),
    absolute_offset: ctx.absolute_byte_offset(),                submatches:
    submatches.as_slice(),            });          self.json.write_message(&msg)?;
    Ok(true)      }}
```

The `matched` callback handles the common case; `context` handles contextual lines which may have matches when using inverted search (`-v` flag).

---

## Section 8: The SubMatches Small-Vector Optimization

```rust
/// SubMatches represents a set of matches in a contiguous range of bytes.
///
/// Optimizes for the common case of exactly one match per line.
enum SubMatches<'a> {
    Empty,
    Small([jsont::SubMatch<'a>; 1]),  // Fixed array - no allocation
    Big(Vec<jsont::SubMatch<'a>>),    // Heap allocation for multiple matches
}

impl<'a> SubMatches<'a> {
    /// Create a new set of match ranges from matches and corresponding bytes.
    fn new(
        bytes: &'a [u8],
        matches: &[Match],
        replacement: Option<(&'a [u8], &'a [Match])>,
    ) -> SubMatches<'a> {
        if matches.len() == 1 {
            // Common case: single match - use fixed-size array
            let mat = matches[0];
            SubMatches::Small([jsont::SubMatch {
                m: &bytes[mat],
                replacement: replacement
                    .map(|(rbuf, rmatches)| &rbuf[rmatches[0]]),
                start: mat.start(),
                end: mat.end(),
            }])
        } else {
            // Rare case: multiple matches - allocate vector
            let mut match_ranges = vec![];
            for (i, &mat) in matches.iter().enumerate() {
                match_ranges.push(jsont::SubMatch {
                    m: &bytes[mat],
                    replacement: replacement
                        .map(|(rbuf, rmatches)| &rbuf[rmatches[i]]),
                    start: mat.start(),
                    end: mat.end(),
                });
            }
            SubMatches::Big(match_ranges)
        }
    }

    /// Create an empty set of match ranges.
    fn empty() -> SubMatches<'static> {
        SubMatches::Empty
    }

    /// Return this set of match ranges as a slice.
    fn as_slice(&self) -> &[jsont::SubMatch<'_>] {
        match *self {
            SubMatches::Empty => &[],
            SubMatches::Small(ref x) => x,
            SubMatches::Big(ref x) => x,
        }
    }
```

```
    }}
```

This enum-based small-vector optimization avoids heap allocation for the most common case (one match per line), while gracefully handling multiple matches when necessary.

## Quick Reference

### Message Types

| Type | Purpose | Key Fields |
| --- | --- | --- |
| `begin` | Search started | `path` |
| `end` | Search finished | `path`, `binary_offset`, `stats` |
| `match` | Match found | `path`, `lines`, `line_number`, `absolute_offset`, `submatches` |
| `context` | Context line | Same as `match` |

### Arbitrary Data Object

```
// Valid UTF-8
{"text": "readable string"}

// Invalid UTF-8
{"bytes": "base64encodeddata=="}
```

### Builder Configuration

| Method | Default | Purpose |
| --- | --- | --- |
| `pretty(bool)` | `false` | Multi-line JSON output |

| | | |
|---|---|---|
| `always_begin_end(bool)` | `false` | Emit begin/end with no matches |
| `replacement(Option<Vec<u8>>)` | `None` | Text substitution pattern |

## Key Type Signatures

```rust
// Create printer
pub fn new(wtr: W) -> JSON<W>

// Create sink for searching
pub fn sink<M: Matcher>(&mut self, matcher: M) -> JSONSink<'static, '_, M, W>
pub fn sink_with_path<M, P>(&mut self, matcher: M, path: &P) -> JSONSink<'_, '_, M, W>

// Sink trait implementation
impl<M: Matcher, W: io::Write> Sink for JSONSink<'_, '_, M, W> {
    type Error = io::Error;
    fn matched(&mut self, ...) -> Result<bool, io::Error>
    fn context(&mut self, ...) -> Result<bool, io::Error>
}
```