

I'll explain each class line by line, excluding package declarations and imports.

ReportFormatters.java - Utility Class for Formatting

```
public final class ReportFormatters {
```

Final class - cannot be extended (subclassed). This is a utility class.

```
    public static final DateTimeFormatter DATE_FORMAT = DateTimeFormatter.ofPattern("dd-MM-yyyy");
```

Static constant date formatter - creates a formatter that converts dates to "dd-MM-yyyy" format (e.g., "25-12-2024"). **static final** means it's shared across all uses and cannot be changed.

```
    private ReportFormatters() {  
        // Utility class - no instantiation  
    }
```

Private constructor - prevents anyone from creating instances of this class (since it only has static methods). This is a standard pattern for utility classes.

```
    /**  
     * Format a LocalDate to dd-MM-yyyy string.  
     * Returns "-" if date is null.  
     */  
    public static String formatDate(LocalDate date) {
```

Static method to format dates - can be called without creating an object (e.g., `ReportFormatters.formatDate(someDate)`).

```
        return date != null ? date.format(DATE_FORMAT) : "-";
```

Ternary operator - if date is not null, format it using DATE_FORMAT, otherwise return "-" (dash). This prevents null pointer exceptions.

```
    /**  
     * Format a currency value with ₹ symbol.  
     * Returns "₹0.00" if value is null.  
     */  
    public static String formatCurrency(Double value) {
```

Static method for currency formatting - adds rupee symbol.

```
        return "₹" + String.format("%.2f", value != null ? value : 0.0);
```

Formats currency - `String.format("%.2f", ...)` formats number with comma separators and 2 decimal places. If value is null, uses 0.0. Prepends ₹ symbol. Example output: "₹1,234.56".

```
    /**  
     * Format a currency value as raw number (for Excel numeric cells).  
     */  
    public static double formatCurrencyRaw(Double value) {
```

Returns raw number - for Excel cells that need actual numbers (not formatted strings).

```
    return value != null ? value : 0.0;
```

Returns value or 0.0 if null - simple null-safe conversion.

AbstractExcelExporter.java - Base Class for Excel Export

```
public abstract class AbstractExcelExporter {
```

Abstract class - cannot be instantiated directly. Must be extended by concrete classes. Provides common functionality for all Excel exporters.

```
    /**
     * Create a bold header style with background color.
     */
    protected CellStyle createHeaderStyle(Workbook workbook) {
```

Protected method - accessible to subclasses. Creates styling for header cells.

```
        CellStyle style = workbook.createCellStyle();
```

Creates new cell style object from the workbook.

```
        Font font = workbook.createFont();
```

Creates font object from workbook.

```
        font.setBold(true);
```

Makes font bold.

```
        style.setFont(font);
```

Applies the bold font to the cell style.

```
        style.setFillForegroundColor(IndexedColors.GREY_25_PERCENT.getIndex());
```

Sets background color to light grey (25% grey).

```
        style.setFillPattern(FillPatternType.SOLID_FOREGROUND);
```

Sets fill pattern to solid - ensures the background color fills the entire cell.

```
        style.setBorderBottom(BorderStyle.THIN);
        style.setBorderTop(BorderStyle.THIN);
        style.setBorderLeft(BorderStyle.THIN);
        style.setBorderRight(BorderStyle.THIN);
```

Adds thin borders to all four sides of the cell.

```
    return style;
}
```

Returns the configured style.

```
    /**
     * Create a date cell style for proper Excel date formatting.
     */
    protected CellStyle createDateCellStyle(Workbook workbook) {
```

Creates style for date cells - makes Excel recognize values as dates.

```
CellStyle style = workbook.createCellStyle();
```

Creates new style.

```
CreationHelper createHelper = workbook.getCreationHelper();
```

Gets creation helper - utility object from workbook for creating formats.

```
style.setDataFormat(createHelper.createDataFormat().getFormat("dd-mm-yyyy"));
```

Sets data format to date - tells Excel to display the cell as a date in dd-mm-yyyy format.

```
return style;
```

```
}
```

```
/**
```

```
* Create a currency cell style.
```

```
*/
```

```
protected CellStyle createCurrencyCellStyle(Workbook workbook) {
```

Creates style for currency cells.

```
CellStyle style = workbook.createCellStyle();
```

```
CreationHelper createHelper = workbook.getCreationHelper();
```

Creates style and gets helper.

```
style.setDataFormat(createHelper.createDataFormat().getFormat("₹#,##0.00"));
```

Sets currency format - displays with ₹ symbol, comma separators, and 2 decimal places. Excel will treat this as a number, not text.

```
return style;
```

```
}
```

```
/**
```

```
* Auto-size all columns in a sheet.
```

```
*/
```

```
protected void autoSizeColumns(Sheet sheet, int columnCount) {
```

Method to auto-size columns - adjusts column widths to fit content.

```
for (int i = 0; i < columnCount; i++) {
```

Loops through each column from 0 to columnCount-1.

```
    sheet.autoSizeColumn(i);
```

Auto-sizes column i - Excel calculates optimal width based on content.

```
}
```

```
}
```

```
/**
```

```
* Create header row with styled cells.
```

```
*/
```

```
protected void createHeaderRow(Sheet sheet, CellStyle headerStyle, String... headers) {
```

Creates header row - String... headers is varargs, allows passing multiple strings.

```
Row row = sheet.createRow(0);
```

Creates row at index 0 (first row).

```
for (int i = 0; i < headers.length; i++) {
```

Loops through each header string.

```
    Cell cell = row.createCell(i);
```

Creates cell at column i.

```
    cell.setCellValue(headers[i]);
```

Sets cell value to the header text.

```
    cell.setCellStyle(headerStyle);
```

Applies header style to the cell.

```
}
```

```
/**
```

```
 * Write workbook to byte array.
```

```
 */
```

```
protected byte[] writeToBytes(Workbook workbook) {
```

Converts workbook to byte array - for sending as HTTP response.

```
    try (ByteArrayOutputStream out = new ByteArrayOutputStream()) {
```

Try-with-resources - creates ByteArrayOutputStream and automatically closes it. Writes to memory instead of file.

```
        workbook.write(out);
```

Writes workbook content to the output stream.

```
    return out.toByteArray();
```

Converts stream to byte array and returns it.

```
    } catch (Exception e) {
        throw new RuntimeException("Excel generation failed", e);
```

Catches any exceptions and wraps in RuntimeException with custom message.



AbstractPdfExporter.java - Base Class for PDF Export

```
public abstract class AbstractPdfExporter {
```

Abstract base class for PDF exporters.

```
    protected static final BaseColor HEADER_BG_BLUE = new BaseColor(52, 152, 219);
```

```
    protected static final BaseColor HEADER_BG_GREEN = new BaseColor(46, 204, 113);
```

```
    protected static final BaseColor HEADER_BG_YELLOW = new BaseColor(241, 196, 15);
```

Predefined colors - RGB values for header backgrounds. `protected static final` means subclasses can access these constant colors.

```
/**  
 * Create title font.  
 */  
protected Font createTitleFont() {  
  
Creates font for document titles.  
  
    return new Font(Font.FontFamily.HELVETICA, 18, Font.BOLD, BaseColor.DARK_GRAY);  
}  
}
```

Returns Helvetica font - size 18, bold, dark gray color.

```
/**  
 * Create header font (white, bold).  
 */  
protected Font createHeaderFont() {  
    return new Font(Font.FontFamily.HELVETICA, 10, Font.BOLD, BaseColor.WHITE);  
}  
}
```

Returns header font - size 10, bold, white (for colored header backgrounds).

```
/**  
 * Create data font.  
 */  
protected Font createDataFont() {  
    return new Font(Font.FontFamily.HELVETICA, 9);  
}  
}
```

Returns data font - size 9, regular (not bold), default color.

```
/**  
 * Create filter info font.  
 */  
protected Font createFilterFont() {  
    return new Font(Font.FontFamily.HELVETICA, 10, Font.ITALIC, BaseColor.GRAY);  
}  
}
```

Returns filter font - size 10, italic, gray - for showing applied filters.

```
/**  
 * Create a styled header cell.  
 */  
protected PdfPCell createHeaderCell(String text, Font font, BaseColor bgColor) {  
}  
}
```

Creates styled table header cell.

```
PdfPCell cell = new PdfPCell(new Phrase(text, font));
```

Creates PDF table cell containing a Phrase (text with font).

```
cell.setBackgroundColor(bgColor);
```

Sets cell background color.

```
cell.setHorizontalAlignment(Element.ALIGN_CENTER);
```

Centers text horizontally in the cell.

```
    cell.setPadding(8);
```

Adds 8 points of padding inside the cell (space between content and borders).

```
    return cell;  
}
```

```
/**  
 * Add title to document.  
 */  
protected void addTitle(Document document, String titleText) throws DocumentException {
```

Adds centered title to PDF document. throws DocumentException declares possible exception.

```
    Paragraph title = new Paragraph(titleText, createTitleFont());
```

Creates paragraph with title text and title font.

```
    title.setAlignment(Element.ALIGN_CENTER);
```

Centers the title.

```
    document.add(title);
```

Adds title to document.

```
    document.add(Chunk.NEWLINE);
```

Adds blank line after title.

```
}
```

```
/**  
 * Get the header background color for this exporter.  
 * Subclasses can override to customize.  
 */  
protected BaseColor getHeaderBackgroundColor() {  
    return HEADER_BG_BLUE;  
}
```

Default header color method - returns blue. Subclasses can override to use different colors (green, yellow, etc.).

ClaimReportExcelExporter.java - Claims Excel Export

```
@Component  
public class ClaimReportExcelExporter extends AbstractExcelExporter {
```

Spring component that extends AbstractExcelExporter. Inherits common Excel functionality.

```
    public byte[] export(List<ClaimReportDto> data) {
```

Export method - takes list of claim DTOs, returns Excel as bytes.

```
        try (Workbook workbook = new XSSFWorkbook()) {
```

Try-with-resources - creates Excel workbook (XLSX format) and auto-closes.

```
            Sheet sheet = workbook.createSheet("Claims Summary");
```

Creates sheet named "Claims Summary".

```
CellStyle headerStyle = createHeaderStyle(workbook);
```

Creates header style using inherited method.

```
CellStyle currencyStyle = createCurrencyCellStyle(workbook);
```

Creates currency style for amount cells.

```
// Header row  
createHeaderRow(sheet, headerStyle,  
    "Enrollment ID", "Claim ID", "Approved Amount", "Claim Date", "Status");
```

Creates header row with 5 columns using inherited method.

```
// Data rows  
int rowIdx = 1;
```

Row index starts at 1 (row 0 is headers).

```
for (ClaimReportDto dto : data) {
```

Loops through each claim DTO.

```
    Row row = sheet.createRow(rowIdx++);
```

Creates new row and increments rowIdx (post-increment: use current value, then add 1).

```
    row.createCell(0).setCellValue(dto.getEnrollmentId());
```

Creates cell in column 0 and sets value to enrollment ID.

```
    Cell claimIdCell = row.createCell(1);  
    if (dto.getClaimId() != null) {  
        claimIdCell.setCellValue(dto.getClaimId());  
    } else {  
        claimIdCell.setCellValue("-");  
    }
```

Creates claim ID cell - if claim ID exists, shows it; otherwise shows "-". Handles null claims.

```
    Cell amountCell = row.createCell(2);  
    amountCell.setCellValue(ReportFormatters.formatCurrencyRaw(dto.getApprovedAmount()));  
    amountCell.setCellStyle(currencyStyle);
```

Creates amount cell - sets numeric value (not formatted string), then applies currency style. Excel will display as ₹1,234.56.

```
    row.createCell(3).setCellValue(ReportFormatters.formatDate(dto.getClaimDate()));
```

Creates date cell with formatted date string.

```
    row.createCell(4).setCellValue(dto.getClaimStatus());
```

Creates status cell with claim status.

```
}
```

```
autoSizeColumns(sheet, 5);
```

Auto-sizes all 5 columns to fit content.

```
return writeToBytes(workbook);
```

Converts workbook to bytes using inherited method.

```
} catch (Exception e) {  
    throw new RuntimeException("Excel generation failed", e);  
}
```

Catches exceptions and wraps in RuntimeException.

ClaimReportPdfExporter.java - Claims PDF Export

```
@Component  
public class ClaimReportPdfExporter extends AbstractPdfExporter {
```

Spring component extending AbstractPdfExporter.

```
    public byte[] export(List<ClaimReportDto> data) {
```

Export method for claims PDF.

```
        Document document = new Document(PageSize.A4.rotate());
```

Creates PDF document - A4 size in landscape orientation (rotated 90 degrees).

```
        ByteArrayOutputStream out = new ByteArrayOutputStream();
```

Creates output stream to write PDF to memory.

```
        try {  
            PdfWriter.getInstance(document, out);
```

Creates PDF writer - connects document to output stream.

```
            document.open();
```

Opens document for writing - must be called before adding content.

```
            addTitle(document, "Claims Summary by Enrollment");
```

Adds title using inherited method.

```
            PdfPTable table = new PdfPTable(5);
```

Creates table with 5 columns.

```
            table.setWidthPercentage(100);
```

Sets table width to 100% of page width.

```
            table.setSpacingBefore(10);
```

Adds 10 points of space above the table.

```
            // Headers  
            Font headerFont = createHeaderFont();  
            BaseColor headerBg = getHeaderBackgroundColor();
```

Gets header font and background color from inherited methods.

```
            String[] headers = {"Enrollment ID", "Claim ID", "Approved Amount", "Claim Date", "Status"};  
            for (String h : headers) {
```

```
    table.addCell(createHeaderCell(h, headerFont, headerBg));
}
```

Creates header cells - loops through header names and adds styled cells to table.

```
// Data
Font dataFont = createDataFont();
```

Gets data font.

```
for (ClaimReportDto dto : data) {
```

Loops through claim data.

```
    table.addCell(new Phrase("#" + dto.getEnrollmentId(), dataFont));
```

Adds enrollment ID cell with "#" prefix.

```
    table.addCell(new Phrase(dto.getClaimId() != null ? String.valueOf(dto.getClaimId()) : "-", dataFont));
```

Adds claim ID cell - converts to String if not null, otherwise "-".

```
    table.addCell(new Phrase(ReportFormatters.formatCurrency(dto.getApprovedAmount()), dataFont));
```

Adds amount cell - formatted with ₹ symbol.

```
    table.addCell(new Phrase(ReportFormatters.formatDate(dto.getClaimDate()), dataFont));
```

Adds date cell - formatted as dd-MM-yyyy.

```
    table.addCell(new Phrase(dto.getClaimStatus(), dataFont));
```

Adds status cell.

```
}
```

```
document.add(table);
```

Adds table to document.

```
document.close();
```

Closes document - finalizes PDF content.

```
} catch (Exception e) {
    throw new RuntimeException("PDF generation failed", e);
}
```

```
return out.toByteArray();
}
```

Returns PDF as byte array.

EmployeeCoverageExcelExporter.java - Employee Coverage Excel Export

```
@Component
```

```
public class EmployeeCoverageExcelExporter extends AbstractExcelExporter {
```

Spring component for exporting employee coverage to Excel.

```
    public byte[] export(List<EmployeeCoverageReportDTO> data, String filters) {
```

Export method - takes employee data and filter description.

```
        try (Workbook workbook = new XSSFWorkbook()) {  
            Sheet sheet = workbook.createSheet("Employee Coverage Report");
```

Creates workbook and sheet.

```
            CellStyle headerStyle = createHeaderStyle(workbook);
```

Creates header style.

```
            // Add filter info row  
            Row filterRow = sheet.createRow(0);
```

Creates first row for filter information.

```
                Cell filterCell = filterRow.createCell(0);  
                filterCell.setCellValue("Applied Filters: " + filters);
```

Creates cell in first row showing which filters were applied.

```
                // Empty row  
                sheet.createRow(1);
```

Creates blank row 1 for spacing.

```
                // Create header row  
                Row header = sheet.createRow(2);
```

Creates header row at index 2 (third row).

```
                String[] columns = { "Employee Code", "Name", "Email", "Designation", "Joining Date",  
                    "Category", "Status", "Enrolled", "Active Policies" };
```

Array of column headers - 9 columns total.

```
                for (int i = 0; i < columns.length; i++) {  
                    Cell cell = header.createCell(i);  
                    cell.setCellValue(columns[i]);  
                    cell.setCellStyle(headerStyle);  
                }
```

Creates and styles header cells for each column.

```
                // Add data rows  
                int rowIdx = 3;
```

Data starts at row 3 (fourth row).

```
                for (EmployeeCoverageReportDTO dto : data) {
```

Loops through employee data.

```
                    Row row = sheet.createRow(rowIdx++);
```

Creates new row and increments counter.

```
                    row.createCell(0).setCellValue(dto.getEmployeeCode());  
                    row.createCell(1).setCellValue(dto.getEmployeeName());
```

```
row.createCell(2).setCellValue(dto.getEmail());  
row.createCell(3).setCellValue(dto.getDesignation());
```

Adds employee basic info to cells 0-3.

```
row.createCell(4).setCellValue(ReportFormatters.formatDate(dto.getJoiningDate()));
```

Adds formatted joining date.

```
row.createCell(5).setCellValue(dto.getCategory().name());
```

Adds category - name() converts enum to String (e.g., "JUNIOR" or "SENIOR").

```
row.createCell(6).setCellValue(dto.getStatus().name());
```

Adds status - enum converted to String (e.g., "ACTIVE", "NOTICE", "EXITED").

```
row.createCell(7).setCellValue(dto.isEnrolled() ? "Yes" : "No");
```

Adds enrollment status - "Yes" if enrolled, "No" otherwise.

```
row.createCell(8).setCellValue(dto.getActiveEnrollmentCount());
```

Adds count of active policies.

```
}
```

```
autoSizeColumns(sheet, columns.length);
```

Auto-sizes all 9 columns.

```
return writeToBytes(workbook);  
} catch (Exception e) {  
    throw new RuntimeException("Excel generation failed", e);  
}
```

EmployeeCoveragePdfExporter.java - Employee Coverage PDF Export

```
@Component  
public class EmployeeCoveragePdfExporter extends AbstractPdfExporter {  
  
    @Override  
    protected BaseColor getHeaderBackgroundColor() {  
        return HEADER_BG_GREEN;  
    }
```

Overrides default header color - uses green instead of blue.

```
public byte[] export(List<EmployeeCoverageReportDTO> data, String filters) {  
    Document document = new Document(PageSize.A4.rotate()); // Landscape for more columns
```

Creates landscape A4 document - comment explains why (needs more width for 8 columns).

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
```

```
try {
    PdfWriter.getInstance(document, ou);
    document.open();

    addTitle(document, "Employee Coverage Report");
```

Standard PDF setup and adds title.

```
// Filters info
Paragraph filterInfo = new Paragraph("Applied Filters: " + filters, createFilterFont());
```

Creates paragraph showing filters with filter font (italic, gray).

```
document.add(filterInfo);
document.add(Chunk.NEWLINE);
```

Adds filter info and blank line to document.

```
// Create table with 8 columns
PdfPTable table = new PdfPTable(8);
```

Creates table with 8 columns (no "Active Policies" column in PDF version).

```
table.setWidthPercentage(100);
table.setSpacingBefore(10f);
```

Sets table width and spacing.

```
// Add headers
Font headerFont = createHeaderFont();
BaseColor headerBg = getHeaderBackgroundColor();
String[] headers = { "Code", "Name", "Email", "Designation", "Joining Date",
    "Category", "Status", "Enrolled" };
for (String h : headers) {
    table.addCell(createHeaderCell(h, headerFont, headerBg));
}
```

Creates header cells with green background.

```
// Data
Font dataFont = createDataFont();
for (EmployeeCoverageReportDTO dto : data) {
```

Gets data font and loops through employees.

```
table.addCell(new Phrase(dto.getEmployeeCode(), dataFont));
table.addCell(new Phrase(dto.getEmployeeName(), dataFont));
table.addCell(new Phrase(dto.getEmail(), dataFont));
```

Adds code, name, email cells.

```
table.addCell(new Phrase(dto.getDesignation() != null ? dto.getDesignation() : "-", dataFont));
```

Adds designation - shows "-" if null.

```
table.addCell(new Phrase(ReportFormatters.formatDate(dto.getJoiningDate()), dataFont));
```

Adds formatted date.

```
table.addCell(new Phrase(dto.getCategory().name(), dataFont));
```

Adds category as String.

```
// Status with color - using enum comparison  
PdfPCell statusCell = new PdfPCell(new Phrase(dto.getStatus().name(), dataFont));
```

Creates status cell with status name.

```
if (dto.getStatus() == EmployeeStatus.ACTIVE) {  
    statusCell.setBackgroundColor(new BaseColor(200, 255, 200));
```

If ACTIVE - sets light green background (RGB: 200, 255, 200).

```
} else if (dto.getStatus() == EmployeeStatus.EXITED) {  
    statusCell.setBackgroundColor(new BaseColor(255, 200, 200));
```

If EXITED - sets light red background.

```
} else {  
    statusCell.setBackgroundColor(new BaseColor(255, 255, 200));
```

Otherwise (NOTICE) - sets light yellow background.

```
}
```

```
table.addCell(statusCell);
```

Adds the colored status cell to table.

```
// Enrollment with color  
PdfPCell enrollCell = new PdfPCell(  
    new Phrase(dto.isEnrolled() ? "Yes (" + dto.getActiveEnrollmentCount() + ")" : "No",  
    dataFont));
```

Creates enrollment cell - shows "Yes (2)" if enrolled with 2 policies, or "No" if not enrolled.

```
enrollCell.setBackgroundColor(dto.isEnrolled() ? new BaseColor(200, 220, 255) :  
    BaseColor.WHITE);
```

Sets background - light blue if enrolled, white if not.

```
table.addCell(enrollCell);  
}
```

Adds enrollment cell.

```
document.add(table);
```

Adds table to document.

```
// Footer with count  
document.add(Chunk.NEWLINE);  
Font footerFont = new Font(Font.FontFamily.HELVETICA, 10, Font.BOLD);  
document.add(new Paragraph("Total Employees: " + data.size(), footerFont));
```

Adds footer showing total employee count in bold.

```
document.close();  
} catch (Exception e) {  
    throw new RuntimeException("PDF generation failed", e);  
}
```

```
return out.toByteArray();  
}
```

EmployeeReportExcelExporter.java - Simple Employee Count Excel

```
@Component
public class EmployeeReportExcelExporter extends AbstractExcelExporter {

    public byte[] export(List<EmployeeReportDto> data) {
        try (Workbook workbook = new XSSFWorkbook()) {
            Sheet sheet = workbook.createSheet("Employees");

            CellStyle headerStyle = createHeaderStyle(workbook);

            // Header row
            createHeaderRow(sheet, headerStyle, "Org ID", "Org Name", "Employee Count");
        }
    }
}
```

Creates simple report with 3 columns.

```
// Data rows
int rowIdx = 1;
for (EmployeeReportDto dto : data) {
    Row row = sheet.createRow(rowIdx++);
    row.createCell(0).setCellValue(dto.getOrganizationId());
    row.createCell(1).setCellValue(dto.getOrganizationName());
    row.createCell(2).setCellValue(dto.getEmployeeCount());
}
```

Adds data rows - organization ID, name, and employee count.

```
autoSizeColumns(sheet, 3);
return writeToBytes(workbook);
} catch (Exception e) {
    throw new RuntimeException("Excel generation failed", e);
}
}
```

EmployeeReportPdfExporter.java - Simple Employee Count PDF

```
@Component
public class EmployeeReportPdfExporter extends AbstractPdfExporter {

    public byte[] export(List<EmployeeReportDto> data) {
        Document document = new Document();
        ByteArrayOutputStream out = new ByteArrayOutputStream();
    }
}
```

Creates document in default portrait A4 (not rotated).

```

try {
    PdfWriter.getInstance(document, out);
    document.open();

    addTitle(document, "Employee Report");

    PdfPTable table = new PdfPTable(3);
    table.setWidthPercentage(100);
    table.setSpacingBefore(10f);

    // Headers
    Font headerFont = createHeaderFont();
    BaseColor headerBg = getHeaderBackgroundColor();
    String[] headers = { "Org ID", "Org Name", "Employee Count" };
    for (String h : headers) {
        table.addCell(createHeaderCell(h, headerFont, headerBg));
    }

    // Data
    Font dataFont = createDataFont();
    for (EmployeeReportDto dto : data) {
        table.addCell(new Phrase(String.valueOf(dto.getOrganizationId()), dataFont));
        table.addCell(new Phrase(dto.getOrganizationName(), dataFont));
        table.addCell(new Phrase(String.valueOf(dto.getEmployeeCount()), dataFont));
    }
}

```

Converts Long to String for display.

```

        table.addCell(new Phrase(dto.getOrganizationName(), dataFont));
        table.addCell(new Phrase(String.valueOf(dto.getEmployeeCount()), dataFont));
}

```

Adds org name and count.

```

    }
    document.add(table);
    document.close();
} catch (Exception e) {
    throw new RuntimeException("PDF generation failed", e);
}

```

```

return out.toByteArray();
}
}

```

PremiumReportExcelExporter.java - Premium Collection Excel

```

@Component
public class PremiumReportExcelExporter extends AbstractExcelExporter {

    public byte[] export(List<PremiumReportDto> data) {
}

```

```

try (Workbook workbook = new XSSFWorkbook()) {
    Sheet sheet = workbook.createSheet("Premium Report");

    CellStyle headerStyle = createHeaderStyle(workbook);
    CellStyle currencyStyle = createCurrencyCellStyle(workbook);

    // Header row
    createHeaderRow(sheet, headerStyle,
        "Organization ID", "Organization Name", "Total Premium Collected");

    // Data rows
    int rowIdx = 1;
    for (PremiumReportDto dto : data) {
        Row row = sheet.createRow(rowIdx++);
        row.createCell(0).setCellValue(dto.getOrganizationId());
        row.createCell(1).setCellValue(dto.getOrganizationName());

        Cell premiumCell = row.createCell(2);

        premiumCell.setCellValue(ReportFormatters.formatCurrencyRaw(dto.getTotalPremiumCollected()));
        premiumCell.setCellStyle(currencyStyle);
    }
}

```

Adds premium amount as numeric value with currency style.

```

    autoSizeColumns(sheet, 3);
    return writeToBytes(workbook);
} catch (Exception e) {
    throw new RuntimeException("Excel generation failed", e);
}
}

```

PremiumReportPdfExporter.java - Premium Collection PDF

```

@Component
public class PremiumReportPdfExporter extends AbstractPdfExporter {

    @Override
    protected BaseColor getHeaderBackgroundColor() {
        return HEADER_BG_YELLOW;
    }
}

```

Overrides to use yellow header background.

```

public byte[] export(List<PremiumReportDto> data) {
    Document document = new Document(PageSize.A4);
}

```

Portrait A4 document (3 columns don't need landscape).

```
ByteArrayOutputStream out = new ByteArrayOutputStream();

try {
    PdfWriter.getInstance(document, out);
    document.open();

    addTitle(document, "Premium Collected by Organization");

    PdfPTable table = new PdfPTable(3);
    table.setWidthPercentage(100);
    table.setSpacingBefore(10f);

    // Headers
    Font headerFont = createHeaderFont();
    BaseColor headerBg = getHeaderBackgroundColor();
```

Gets yellow header background.

```
String[] headers = {"Organization ID", "Organization Name", "Total Premium Collected"};
for (String h : headers) {
    table.addCell(createHeaderCell(h, headerFont, headerBg));
}

// Data
Font dataFont = createDataFont();
for (PremiumReportDto dto : data) {
    table.addCell(new Phrase(String.valueOf(dto.getOrganizationId()), dataFont));
    table.addCell(new Phrase(dto.getOrganizationName(), dataFont));
    table.addCell(new Phrase(ReportFormatters.formatCurrency(dto.getTotalPremiumCollected()), dataFont));
```

Adds premium as formatted string with ₹ symbol.

```
}
```

```
document.add(table);
document.close();
} catch (Exception e) {
    throw new RuntimeException("PDF generation failed", e);
}

return out.toByteArray();
}
```

ReportService.java - Service Interface

```
public interface ReportService {
```

Interface defining report methods - implementations provide the actual logic.

```
/**  
 * Get employee count grouped by organization.  
 *  
 * @param organizationId filter by org ID, or null for all  
 */  
List<EmployeeReportDto> getEmployeeCountByOrganization(Long organizationId);
```

Method signature for employee count report - takes optional org ID filter.

```
/**  
 * Get claim summary grouped by enrollment.  
 *  
 * @param status filter by claim status, or null/ALL for all  
 */  
List<ClaimReportDto> getClaimSummaryByEnrollment(String status);
```

Method for claim summary - takes optional status filter.

```
/**  
 * Get total premium collected grouped by organization.  
 *  
 * @param organizationId filter by org ID, or null for all  
 */  
List<PremiumReportDto> getPremiumCollectedByOrganization(Long organizationId);
```

Method for premium report - takes optional org ID filter.

ReportServiceImpl.java - Service Implementation

```
@Service  
public class ReportServiceImpl implements ReportService {
```

Spring service that implements ReportService interface.

```
@PersistenceContext  
private EntityManager em;
```

Injects EntityManager - JPA object for executing custom queries. **@PersistenceContext** is JPA's way of dependency injection.

```
@Override  
public List<EmployeeReportDto> getEmployeeCountByOrganization(Long organizationId) {
```

Implements interface method.

```
String query = "";  
query += "SELECT new EmployeeReportDto(  
    o.organizationId,  
    o.organizationName,  
    COUNT(e))";  
query += "FROM Organization o";
```

```
        LEFT JOIN o.employee e
        WHERE (:orgId IS NULL OR o.organizationId = :orgId)
        GROUP BY o.organizationId, o.organizationName
```

JPQL query using text block (triple quotes). Breakdown:

- SELECT new EmployeeReportDto(...) - creates DTO objects directly from query results
- FROM Organization o - starts from Organization table
- LEFT JOIN o.employee e - joins employees (LEFT JOIN includes orgs with no employees)
- WHERE (:orgId IS NULL OR o.organizationId = :orgId) - if orgId parameter is null, get all orgs; otherwise filter by orgId
- COUNT(e) - counts employees per organization
- GROUP BY - groups results by organization

```
    return em.createQuery(query, EmployeeReportDto.class)
```

Creates typed query - expects EmployeeReportDto results.

```
.setParameter("orgId", organizationId)
```

Sets parameter value - replaces :orgId placeholder.

```
.getResultList()
```

Executes query and returns list.

```
@Override
public List<ClaimReportDto> getClaimSummaryByEnrollment(String status) {
    String query = """
        SELECT new com.employeeinsurancemanagement.dto.ClaimReportDto(
            e.enrollmentId,
            c.claimId,
            COALESCE(c.approvedAmount, 0.0),
            c.claimDate,
            c.claimStatus
        )
        FROM Enrollment e
        LEFT JOIN e.claims c
        WHERE (:status IS NULL OR c.claimStatus = :status)
        ORDER BY c.claimDate DESC
    """
```

JPQL for claim summary:

- COALESCE(c.approvedAmount, 0.0) - returns 0.0 if approvedAmount is null
- LEFT JOIN e.claims c - includes enrollments even without claims
- WHERE (:status IS NULL OR c.claimStatus = :status) - optional status filter
- ORDER BY c.claimDate DESC - newest claims first

```
var q = em.createQuery(query, ClaimReportDto.class);
```

Creates query - var is type inference (compiler determines type).

```
if (status != null && !status.equals("ALL")) {
```

Checks if status filter provided and not "ALL".

```
try {
    com.employeeinsurancemanagement.model.ClaimStatus statusEnum =
com.employeeinsurancemanagement.model.ClaimStatus
    .valueOf(status.toUpperCase());
```

Converts string to enum - e.g., "submitted" becomes ClaimStatus.SUBMITTED.

```
q.setParameter("status", statusEnum);
} catch (IllegalArgumentException e) {
    q.setParameter("status", null);
```

If conversion fails (invalid status string), set parameter to null (no filter).

```
}
```

```
} else {
    q.setParameter("status", null);
```

If status is null or "ALL", set parameter to null.

```
}
```

```

    return q.getResultList();
}
```

```
@Override
public List<PremiumReportDto> getPremiumCollectedByOrganization(Long organizationId) {
    String query = """
        SELECT new com.employeeinsurancemanagement.dto.PremiumReportDto(
            o.organizationId,
            o.organizationName,
            COALESCE(SUM(e.premiumAmount), 0)
        )
        FROM Organization o
        LEFT JOIN o.employee emp
        LEFT JOIN emp.enrollments e
        WHERE (:orgId IS NULL OR o.organizationId = :orgId)
        GROUP BY o.organizationId, o.organizationName
        """;
}
```

JPQL for premium collection:

- LEFT JOIN o.employee emp - joins employees
- LEFT JOIN emp.enrollments e - joins enrollments (through employees)
- COALESCE(SUM(e.premiumAmount), 0) - sums premium amounts, returns 0 if null
- GROUP BY - one row per organization

```
return em.createQuery(query, PremiumReportDto.class)
    .setParameter("orgId", organizationId)
    .getResultList();
```



Sets parameter and returns results.

HrReportService.java - HR-Specific Reports

```
@Service  
@RequiredArgsConstructor  
public class HrReportService {
```

Service for HR reports - `@RequiredArgsConstructor` generates constructor for final fields.

```
private final EmployeeRepository employeeRepository;  
private final EnrollmentRepository enrollmentRepository;
```

Injects repositories via constructor.

```
// Whitelist of sortable fields  
private static final Map<String, String> SORT_FIELD_MAP = Map.of(  
    "name", "employeeName",  
    "joiningDate", "joiningDate",  
    "status", "status",  
    "category", "category");
```

Maps sort parameter names to DTO field names - prevents SQL injection by only allowing these fields.

`Map.of()` creates immutable map.

```
/**  
 * Get filtered, sorted, paginated employee coverage report.  
 * ...  
 */  
public EmployeeCoverageReportResult getEmployeeCoverageReport(  
    Long organizationId,  
    String statusFilter,  
    String categoryFilter,  
    EnrollmentStateFilter enrollmentStateFilter,  
    String sortBy,  
    String sortDir,  
    int page,  
    int pageSize) {
```

Main report method with all filter/sort/pagination parameters.

```
// 1. Fetch all employees for organization  
List<Employee> employees = employeeRepository.findByOrganizationOrganizationId(organizationId);
```

Gets all employees for the organization from database.

```
// 2. Map to DTOs with resolved category and enrollment count  
List<EmployeeCoverageReportDTO> dtos = employees.stream()
```

Starts stream to transform employees to DTOs.

```
.map(emp -> {
```

Maps each employee to a DTO.

```
Employee.EmployeeCategory resolvedCategory = resolveCategory(emp);
```

Resolves category - JUNIOR or SENIOR based on tenure.

```
int activeEnrollments = enrollmentRepository.countByEmployeeAndEnrollmentStatus(  
    emp, EnrollmentStatus.ACTIVE);
```

Counts active enrollments for this employee.

```
return EmployeeCoverageReportDTO.fromEmployee(emp, resolvedCategory,  
    activeEnrollments);
```

Creates DTO with employee data, resolved category, and enrollment count.

```
})  
.collect(Collectors.toList());
```

Collects to list.

```
// 3. Apply status filter  
if (statusFilter != null && !statusFilter.isEmpty() && !statusFilter.equalsIgnoreCase("ALL")) {
```

Checks if status filter applied - not null, not empty, not "ALL".

```
try {  
    EmployeeStatus status = EmployeeStatus.valueOf(statusFilter.toUpperCase());
```

Converts string to enum - e.g., "active" ↳ EmployeeStatus.ACTIVE.

```
dtos = dtos.stream()  
.filter(dto -> dto.getStatus() == status)  
.collect(Collectors.toList());
```

Filters DTOs to only those matching the status.

```
} catch (IllegalArgumentException ignored) {  
    // Invalid status - ignore filter  
}
```

If invalid status, silently ignore (don't filter).

```
}
```

```
// 4. Apply category filter  
if (categoryFilter != null && !categoryFilter.isEmpty() && !categoryFilter.equalsIgnoreCase("ALL")) {  
    try {  
        Employee.EmployeeCategory category =  
            Employee.EmployeeCategory.valueOf(categoryFilter.toUpperCase());  
        dtos = dtos.stream()  
.filter(dto -> dto.getCategory() == category)  
.collect(Collectors.toList());  
    } catch (IllegalArgumentException ignored) {  
        // Invalid category - ignore filter  
    }  
}
```

Same pattern for category filter - JUNIOR or SENIOR.

```
// 5. Apply enrollment state filter (in-memory, NOT in repository)
```

```
    if (enrollmentStateFilter != null && enrollmentStateFilter != EnrollmentStateFilter.ALL) {
```

Checks enrollment filter - not null and not ALL.

```
        boolean filterEnrolled = enrollmentStateFilter == EnrollmentStateFilter.ENROLLED;
```

Determines filter value - true if ENROLLED, false if NOT_ENROLLED.

```
        dtos = dtos.stream()
            .filter(dto -> dto.isEnrolled() == filterEnrolled)
            .collect(Collectors.toList());
```

Filters by enrollment status - keeps only matching employees.

```
}
```

```
// 6. Apply sorting (whitelisted fields only)
```

```
Comparator<EmployeeCoverageReportDTO> comparator = getComparator(sortBy, sortDir);
```

Gets comparator for sorting based on sort field and direction.

```
        dtos.sort(comparator);
```

Sorts the list using the comparator.

```
// 7. Calculate pagination
```

```
int totalElements = dtos.size();
```

Total number of filtered results.

```
int totalPages = (int) Math.ceil((double) totalElements / pageSize);
```

Calculates total pages - divides total by page size, rounds up. E.g., 23 items / 10 per page = 2.3 → 3 pages.

```
int fromIndex = Math.min(page * pageSize, totalElements);
```

Calculates start index for this page. E.g., page 2, size 10 = index 20. Math.min prevents index overflow.

```
int toIndex = Math.min(fromIndex + pageSize, totalElements);
```

Calculates end index - adds page size, but caps at total elements.

```
List<EmployeeCoverageReportDTO> pageContent = dtos.subList(fromIndex, toIndex);
```

Extracts page slice from the full list. E.g., subList(20, 30) gets items 20-29.

```
    return new EmployeeCoverageReportResult(
        pageContent,
        page,
        pageSize,
        totalElements,
        totalPages);
}
```

Returns result record with page data and metadata.

```
/**  
 * Resolve employee category based on tenure (5+ years = SENIOR).  
 * Same logic as in other services.  
 */
```

```
private Employee.EmployeeCategory resolveCategory(Employee employee) {
```

Helper method to determine category.

```
    if (employee.getJoiningDate() == null) {
        return Employee.EmployeeCategory.JUNIOR;
    }
}
```

If no joining date, default to JUNIOR.

```
    int yearsOfService = Period.between(employee.getJoiningDate(), LocalDate.now()).getYears();
```

Calculates years of service - Period.between finds difference between joining date and today, .getYears() extracts year component.

```
    return yearsOfService >= 5 ? Employee.EmployeeCategory.SENIOR :
        Employee.EmployeeCategory.JUNIOR;
}
```

Returns SENIOR if 5+ years, otherwise JUNIOR.

```
/** 
 * Get comparator for sorting based on whitelisted field.
 * Defaults to name ascending if field is not whitelisted.
 */
private Comparator<EmployeeCoverageReportDTO> getComparator(String sortBy, String sortDir) {
```

Creates comparator for sorting.

```
    boolean ascending = !"desc".equalsIgnoreCase(sortDir);
```

Determines sort direction - ascending unless sortDir is "desc".

```
    // Validate sort field - default to name if not whitelisted
    String field = SORT_FIELD_MAP.getOrDefault(sortBy, "employeeName");
```

Gets validated field name from whitelist - defaults to "employeeName" if not found (prevents SQL injection).

```
    Comparator<EmployeeCoverageReportDTO> comparator;

    switch (field) {
        case "joiningDate":
            comparator = Comparator.comparing(
                EmployeeCoverageReportDTO::getJoiningDate,
                Comparator.nullsLast(Comparator.naturalOrder()));
            break;
```

Comparator for joining date - nullsLast puts null dates at end, naturalOrder sorts dates chronologically.

```
        case "status":
            comparator = Comparator.comparing(
                dto -> dto.getStatus().name());
            break;
```

Comparator for status - compares enum names alphabetically (ACTIVE, EXITED, NOTICE).

```
        case "category":
            comparator = Comparator.comparing(
                dto -> dto.getCategory().name());
            break;
```

Comparator for category - compares JUNIOR vs SENIOR alphabetically.

```
        case "employeeName":  
        default:  
            comparator = Comparator.comparing(  
                EmployeeCoverageReportDTO::getEmployeeName,  
                Comparator.nullsLast(String.CASE_INSENSITIVE_ORDER));  
            break;  
    }  
}
```

Comparator for name (default) - CASE_INSENSITIVE_ORDER ignores case (A = a), nullsLast handles nulls.

```
    return ascending ? comparator : comparator.reversed();  
}
```

Returns comparator - reversed if descending.

```
/**  
 * Result wrapper for paginated report.  
 */  
public record EmployeeCoverageReportResult(  
    List<EmployeeCoverageReportDTO> content,  
    int currentPage,  
    int pageSize,  
    int totalElements,  
    int totalPages) {
```

Java record - immutable data class with automatic constructor, getters, equals, hashCode, toString.

```
    public boolean hasNext() {  
        return currentPage < totalPages - 1;  
    }
```

Checks if there's a next page - e.g., if on page 1 of 3 pages (currentPage=1, totalPages=3), then 1 < 2, so true.

```
    public boolean hasPrevious() {  
        return currentPage > 0;  
    }
```

Checks if there's a previous page - page 0 has no previous, page 1+ has previous.

Summary

This code implements a **comprehensive reporting system** for an employee insurance application:

1. **ReportFormatters** - Centralized date and currency formatting
2. **AbstractExcelExporter** - Base class with common Excel styling/functionality
3. **AbstractPdfExporter** - Base class with common PDF styling/functionality
4. **Multiple Exporters** - Specific exporters for claims, employee coverage, employee counts, and premium reports in both Excel and PDF formats

5. **ReportService** - Interface for data retrieval using JPQL queries

6. **ReportServiceImpl** - Implements queries with EntityManager

7. **HrReportService** - Advanced filtering, sorting, and pagination for employee coverage reports

The architecture follows DRY (Don't Repeat Yourself) principles by using inheritance and utility classes to avoid code duplication.