Prompts

JEP 467: Markdown Documentation Comments

Enable JavaDoc documentation comments to be written in Markdown rather than solely in a mixture of HTML and JavaDoc @-tags.

Goals

- Make API documentation comments easier to write and easier to read in source form by introducing the ability to use Markdown syntax in documentation comments, alongside HTML elements and JavaDoc tags.
- Do not adversely affect the interpretation of existing documentation comments.
- Extend the Compiler Tree API to enable other tools that analyze documentation comments to handle Markdown content in those comments.

Non-Goals

- It is not a goal to enable automated conversion of existing documentation comments to Markdown syntax.

Motivation

Documentation comments are stylized comments appearing in source code, near to the declarations that they serve to document. Documentation comments in Java source code use a combination of HTML and custom JavaDoc tags to mark up the text.

The choice of HTML for a markup language was reasonable in 1995. HTML is powerful, standardized, and was very popular at the time. But while it is no less popular today as a markup language consumed by web browsers, in the years since 1995 HTML has become much less popular as markup that is manually produced by humans because it is tedious to write and hard to read. These days it is more commonly generated from some other markup language that is more suitable for humans. Because HTML is tedious to write, nicely-formatted documentation comments are also tedious to write, and even more tedious since many new developers are not fluent in HTML due to its decline as a human-produced format.

Inline JavaDoc tags, such as {@link} and {@code}, are also cumbersome and are even less familiar to developers, often requiring the author to consult the documentation for their usage. A recent analysis of the documentation comments in the JDK source code showed that over 95% of the uses of inline tags were for code fragments and links to elsewhere in the documentation, suggesting that simpler forms of these constructs would be welcome.

Markdown is a popular markup language for simple documents that is easy to read, easy to write, and easily transformed into HTML. Documentation comments are typically not complicated structured documents, and for the constructs that typically appear in documentation comments, such as paragraphs, lists, styled text, and links, Markdown provides simpler forms than HTML. For those constructs that Markdown does not directly support, Markdown allows the use of HTML as well.

Introducing the ability to use Markdown in documentation comments would bring together the best of both worlds. It would allow concise syntax for the most common constructs and reduce the need for HTML markup and JavaDoc tags, while retaining the ability to use specialized tags for features not available in Markdown. It would make it easier to write and easier to read documentation comments in source code, while retaining the ability to generate the same sort of generated API documentation as before.

Description

As an example of the use of Markdown in a documentation comment, consider the comment for java.lang.Object.hashCode:

```
/**
 * Returns a hash code value for the object. This method is
 * supported for the benefit of hash tables such as those provided by
 * {@link java.util.HashMap}.
 * <p>
 * The general contract of {@code hashCode} is:
 * <ul>
 * <li>Whenever it is invoked on the same object more than once during
 *     an execution of a Java application, the {@code hashCode} method
 *     must consistently return the same integer, provided no information
 *     used in {@code equals} comparisons on the object is modified.
 *     This integer need not remain consistent from one execution of an
 *     application to another execution of the same application.
 * <li>If two objects are equal according to the {@link
 *     #equals(Object) equals} method, then calling the {@code
 *     hashCode} method on each of the two objects must produce the
 *     same integer result.
 * <li>It is <em>not</em> required that if two objects are unequal
 *     according to the {@link #equals(Object) equals} method, then
 *     calling the {@code hashCode} method on each of the two objects
 *     must produce distinct integer results.  However, the programmer
 *     should be aware that producing distinct integer results for
 *     unequal objects may improve the performance of hash tables.
 * </ul>
 *
 * @implSpec
 * As far as is reasonably practical, the {@code hashCode} method defined
 * by class {@code Object} returns distinct integers for distinct objects.
 *
 * @return  a hash code value for this object.
 * @see     java.lang.Object#equals(java.lang.Object)
 * @see     java.lang.System#identityHashCode
```

```
 */
```

The same comment can be written by expressing its structure and styling in Markdown, with no use of HTML and just a few JavaDoc inline tags:

```
/// Returns a hash code value for the object. This method is
/// supported for the benefit of hash tables such as those provided by
/// [java.util.HashMap].
///
/// The general contract of `hashCode` is:
///
///   - Whenever it is invoked on the same object more than once during
///     an execution of a Java application, the `hashCode` method
///     must consistently return the same integer, provided no information
///     used in `equals` comparisons on the object is modified.
///     This integer need not remain consistent from one execution of an
///     application to another execution of the same application.
///   - If two objects are equal according to the
///     [equals][#equals(Object)] method, then calling the
///     `hashCode` method on each of the two objects must produce the
///     same integer result.
///   - It is _not_ required that if two objects are unequal
///     according to the [equals][#equals(Object)] method, then
///     calling the `hashCode` method on each of the two objects
///     must produce distinct integer results.  However, the programmer
///     should be aware that producing distinct integer results for
///     unequal objects may improve the performance of hash tables.
///
/// @implSpec
/// As far as is reasonably practical, the `hashCode` method defined
/// by class `Object` returns distinct integers for distinct objects.
///
/// @return  a hash code value for this object.
/// @see     java.lang.Object#equals(java.lang.Object)
/// @see     java.lang.System#identityHashCode
```

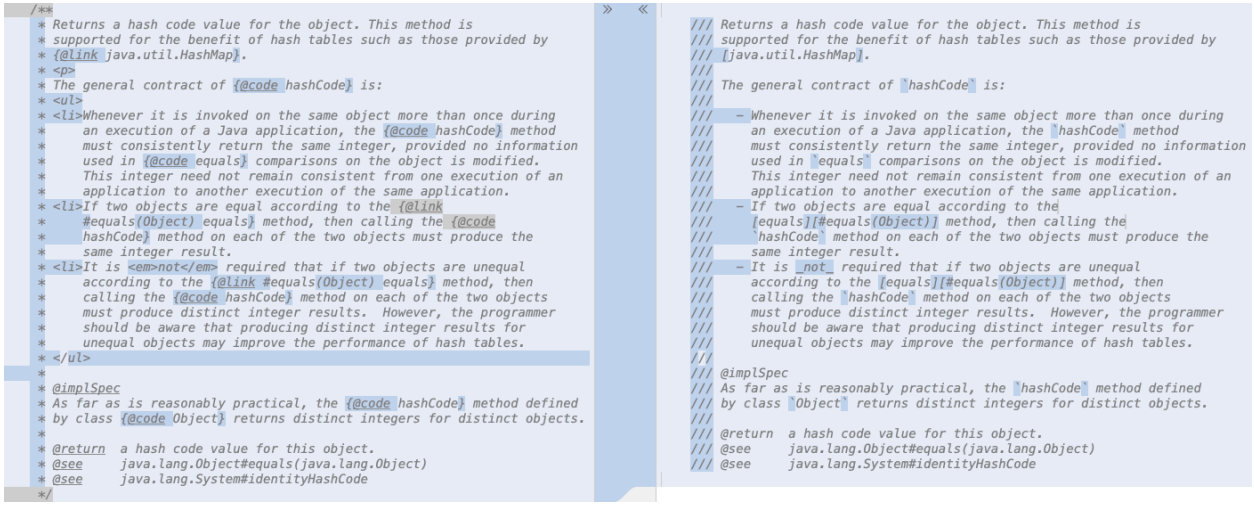
(For the purpose of this example, cosmetic changes such as reflowing the text are deliberately avoided, to aid in before-and-after comparison.)

Key differences to observe:

- The use of Markdown is indicated by a new form of documentation comment in which each line begins with `///` instead of the traditional `/** ... */` syntax.
- The HTML `<p>` element is not required; a blank line indicates a paragraph break.

- The HTML <ul> and <li> elements are replaced by Markdown bullet-list markers, using - to indicate the beginning of each item in the list.
- The HTML element is replaced by using underscores (_) to indicate the font change.
- Instances of the {@code ...} tag are replaced by backticks (`...`) to indicate the monospace font.
- Instances of {@link ...} to link to other program elements are replaced by extended forms of Markdown reference links.
- Instances of block tags, such as @implSpec, @return, and @see, are generally unaffected except that the content of these tags is now also in Markdown, for example here in the backticks of the content of the @implSpec tag.

Here is a screenshot highlighting the differences between the two versions, side by side:



```
/**
 * Returns a hash code value for the object. This method is
 * supported for the benefit of hash tables such as those provided by
 * {@link java.util.HashMap}.
 * <p>
 * The general contract of {@code hashCode} is:
 * <ul>
 * <li>Whenever it is invoked on the same object more than once during
 *     an execution of a Java application, the {@code hashCode} method
 *     must consistently return the same integer, provided no information
 *     used in {@code equals} comparisons on the object is modified.
 *     This integer need not remain consistent from one execution of an
 *     application to another execution of the same application.
 * <li>If two objects are equal according to the {@link
 *     #equals(Object) equals} method, then calling the {@code
 *     hashCode} method on each of the two objects must produce the
 *     same integer result.
 * <li>It is <em>not</em> required that if two objects are unequal
 *     according to the {@link #equals(Object) equals} method, then
 *     calling the {@code hashCode} method on each of the two objects
 *     must produce distinct integer results.  However, the programmer
 *     should be aware that producing distinct integer results for
 *     unequal objects may improve the performance of hash tables.
 * </ul>
 *
 * @implSpec
 * As far as is reasonably practical, the {@code hashCode} method defined
 * by class {@code Object} returns distinct integers for distinct objects.
 *
 * @return  a hash code value for this object.
 * @see     java.lang.Object#equals(java.lang.Object)
 * @see     java.lang.System#identityHashCode
 */
```

```
/// Returns a hash code value for the object. This method is
/// supported for the benefit of hash tables such as those provided by
/// [java.util.HashMap].
///
/// The general contract of `hashCode` is:
///
///   - Whenever it is invoked on the same object more than once during
///     an execution of a Java application, the `hashCode` method
///     must consistently return the same integer, provided no information
///     used in `equals` comparisons on the object is modified.
///     This integer need not remain consistent from one execution of an
///     application to another execution of the same application.
///   - If two objects are equal according to the
///     [equals][#equals(Object)] method, then calling the
///     `hashCode` method on each of the two objects must produce the
///     same integer result.
///   - It is _not_ required that if two objects are unequal
///     according to the [equals][#equals(Object)] method, then
///     calling the `hashCode` method on each of the two objects
///     must produce distinct integer results.  However, the programmer
///     should be aware that producing distinct integer results for
///     unequal objects may improve the performance of hash tables.
///
/// @implSpec
/// As far as is reasonably practical, the `hashCode` method defined
/// by class `Object` returns distinct integers for distinct objects.
///
/// @return  a hash code value for this object.
/// @see     java.lang.Object#equals(java.lang.Object)
/// @see     java.lang.System#identityHashCode
```

Using /// for Markdown documentation comments

We use /// for Markdown comments in order to overcome two issues with traditional /** comments.

- A block comment beginning with /* cannot contain the character sequence */ (JLS §3.7). It is becoming increasingly common to put examples of code in documentation comments. This restriction precludes examples containing embedded /*...*/ comments, or expressions containing the characters */, without the use of disruptive workarounds.
  In // comments, there is no restriction on the characters that may appear on the rest of the line.
- In a traditional documentation comment, beginning with /**, the use of leading whitespace followed by one or more asterisks on each line is optional. When such asterisks are omitted from the lines of a comment there is an ambiguity with Markdown constructs that themselves begin with an asterisk, such as emphases, list items, and thematic breaks.
  In /// comments, there is never any such ambiguity.

It is not an option to change the syntax of the Java language to allow new forms of comment. Therefore, any new style of documentation comment must be in the form of either a traditional /* ... */ block comment or a series of // end-of-line comments.

The above points justify the use of end-of-line comments instead of traditional comments, but the question remains of how to distinguish documentation comments from other end-of-line comments. We use an additional /, which echoes the use of an additional * at the start of traditional documentation comments. Moreover, while not a primary consideration, other languages that support end-of-line documentation comments, such as C#, Dart, and Rust, have successfully used /// for documentation comments for some time now.

Syntax

Markdown documentation comments are written in the CommonMark variant of Markdown. Enhancements to links allow convenient linking to other program elements. Simple GFM pipe tables are supported, as are all JavaDoc tags.

Links

You can create a link to an element declared elsewhere in your API by using an extended form of Markdown reference link, in which the label for the reference is derived from a standard JavaDoc reference to the element itself.

To create a simple link whose text is derived from the identity of the element, simply enclose a reference to the element in square brackets. For example, to link to java.util.List, you can write [java.util.List], or just [List] if there is an import statement for java.util.List in the code. The text of the link will be displayed in the monospace font. The link is equivalent to using the standard JavaDoc {@link ...} tag.

You can link to any kind of program element:

/// - a module [java.base/]
/// - a package [java.util]
/// - a class [String]
/// - a field [String#CASE_INSENSITIVE_ORDER]

/// - a method [String#chars()]

To create a link with alternative text, use the form [text][element]. For example, to create a link to java.util.List with the text a list, you can write [a list][List]. The link will be displayed in the current font, although you can use formatting markup within the text. The link is equivalent to using the standard JavaDoc {@linkplain ...} tag.

For example:

/// - [the `java.base` module][java.base/]
/// - [the `java.util` package][java.util]
/// - [a class][String]
/// - [a field][String#CASE_INSENSITIVE_ORDER]

/// - [a method][String#chars()]

In reference links, you must escape any use of square brackets. This might occur in a reference to a method with an array parameter; for example, you would write a link to String.copyValueOf(char[]) as [String#copyValueOf(char\[\])].

You can use all other forms of Markdown links, including links to URLs, but links to other program elements are likely to be the most common.

Tables

Simple tables are supported, using the syntax of GitHub Flavored Markdown. For example:

```
/// | Latin | Greek |
/// |-------|-------|
/// | a     | alpha |
/// | b     | beta  |
/// | c     | gamma |
```

Captions and other features that may be required for accessibility are not supported. In such situations, the use of HTML tables is still recommended.

JavaDoc tags

JavaDoc tags, both inline tags such as {@inheritDoc} and block tags such as @param and @return, may be used in Markdown documentation comments:

```
/// {@inheritDoc}
/// In addition, this methods calls [#wait()].
///
/// @param i the index
public void m(int i) ...
```

JavaDoc tags may not be used within literal text, such as code spans (`...`) or code blocks, that is, blocks of text that are either indented or enclosed within *fences* such as ``` or ~~~. In other words, the character sequences @... and {@...} have no special meaning within code spans and code blocks:

```
/// The following code span contains literal text, and not a JavaDoc tag:
/// `{@inheritDoc}`
///
/// In the following indented code block, `@Override` is an annotation,
/// and not a JavaDoc tag:
///
///     @Override
///     public void m() ...
///
/// Likewise, in the following fenced code block, `@Override` is an annotation,
/// and not a JavaDoc tag:
///
```

```
/// ```
/// @Override
/// public void m() ...
/// ```
```

For those tags that may contain text with markup, in a Markdown documentation comment that markup is also in Markdown:

```
/// @param l   the list, or `null` if no list is available
```

The {@inheritDoc} tag incorporates documentation for a method from one or more supertypes. The format of the comment containing the tag does not need to be the same as the format of the comment containing the documentation to be inherited:

```
interface Base {
   /** A method. */
   void m()
}

class Derived implements Base {
   /// {@inheritDoc}
   public void m() { }
}
```

User-defined JavaDoc tags may be used in Markdown documentation comments. For example, in the JDK documentation we define and use {@jls ...} as a short form for links to the Java Language Specification, and block tags such as @implSpec and @implNote to introduce sections of particular information:

```
/// For more information on comments, see {@jls 3.7 Comments}.
///
/// @implSpec
/// This implementation does nothing.
public void doSomething() { }
```

Standalone Markdown files

Markdown files in doc-files subdirectories are processed appropriately, in a similar manner to HTML files in such directories. JavaDoc tags in such files are processed. The page title is inferred from the first heading. YAML metadata, such as that supported by the Pandoc Markdown processor, is not supported.

The file containing the content for the generated top-level overview page may also be a Markdown file.

Syntax highlighting and embedded languages

The opening fence in a fenced code block may be followed by an *info string*. The first word of the info string is used to derive the CSS class name in the corresponding generated HTML, and

may also be used by JavaScript libraries to enable syntax highlighting (such as with Prism) and rendering diagrams (such as with Mermaid).

For example, in conjunction with the appropriate libraries, this would display a fragment of CSS code with syntax highlighting:

```
/// ```css
/// p { color: red }
/// ```
```

You can add JavaScript libraries to your documentation by using the javadoc --add-script option.

Syntactical details

Because horizontal whitespace at the beginning and end of each line of Markdown text may be significant, the content of a Markdown documentation comment is determined as follows:

- Any leading whitespace and the three initial / characters are removed from each line.
- The lines are shifted left, by removing leading whitespace characters, until the non-blank line with the least leading whitespace has no remaining leading whitespace.
- Additional leading whitespace and any trailing whitespace in each line is preserved, because it may be significant. For example, whitespace at the beginning of a line may indicate an indented code block or the continuation of a list item, and whitespace at the end of a line may indicate a hard line break.

(The policy to remove leading incidental whitespace is similar to that for String.stripIndent(), except that there is no need to handle trailing blank lines.)

There are no restrictions on the characters that may appear after the /// on each line of the comment. In particular, the comment may contain code samples which may contain comments of their own:

```
/// Here is an example:
///
/// ```
/// /** Hello World! */
/// public class HelloWorld {
///     public static void main(String... args) {
///         System.out.println("Hello World!"); // the traditional example
///     }
/// }
/// ```
```

As well as serving to visually distinguish the new kind of documentation comment, the use of end-of-line (///) comments eliminates the restrictions on the content of the comment that are inherent with the use of traditional (/* ... */) comments. In particular, it is not possible to use the character sequence */ within a traditional comment (JLS §3.7) although it may be desirable to

do so when writing example code containing traditional comments, strings containing glob expressions, and strings containing regular expressions.

For a blank line to be included in the comment, it must begin with any optional whitespace and then ///:

```
/// This is an example ...
///

/// ... of a 3-line comment containing a blank line.
```

A completely blank line will cause any preceding and following comment to be treated as separate comments. In that case, all but the last comment will be discarded, and only the last comment will be considered as a documentation comment for any declaration that may follow:

```
/// This comment will be treated as a "dangling comment" and will be ignored.

/// This is the comment for the following declaration.
public void m() { }
```

The same is true for any other comment not beginning with /// that may appear between two /// comments.

API and implementation

Parsed documentation comments are represented by elements of the com.sun.source.doctree package in the Compiler Tree API.

We introduce a new type of tree node, RawTextTree, which contains uninterpreted text, together with a new tree-node kind, DocTree.Kind.MARKDOWN, which indicates Markdown content in a RawTextTree. We add corresponding new visitRawText methods to DocTreeVisitor and its subtypes, DocTreeScanner and DocTreePathScanner.

RawTextTree nodes with a kind of MARKDOWN represent Markdown content, including HTML constructs but excluding any JavaDoc tags such as {@inheritDoc} and @param.

Markdown text is processed in two phases:

1. *Parsing* — Markdown comments are parsed into a sequence of RawTextTree nodes, each with a kind of DocTree.Kind.MARKDOWN and containing Markdown content, interspersed with standard DocTree nodes for inline and block tags. The inline and block tags are parsed in the same way as for traditional documentation comments, except that tag content is also parsed as Markdown. The sequence of nodes is stored in a DocCommentTree node, in the normal manner.
   Unlike a traditional documentation comment, HTML constructs are not parsed into corresponding DocTree nodes, because too much of the surrounding context needs to be taken into account.
   The Markdown content in the DocCommentTree resulting from the initial parse is then examined for any reference links with no associated link reference definition, and for which the link label syntactically matches a reference to a program element.

Any such link is replaced by an equivalent node representing either {@link ...} or {@linkplain ...}.

2. *Rendering* — The DocCommentTree is rendered by the javadoc tool into HTML that is suitable for inclusion in the page being generated.
   Any sequence of RawTextTree nodes and other nodes is converted into a single string containing the text of the RawTextTree nodes with the Unicode OBJECT REPLACEMENT CHARACTER (U+FFFC) standing in for non-Markdown content. The resulting string is rendered by the Markdown processor and then the U+FFFC characters are replaced in the resulting output by the rendered forms of the non-Markdown content nodes.
   While most of the rendering is straightforward, special attention is given to Markdown headings:
   - The heading level is adjusted according to the enclosing context. This applies whether the heading was initially written in the documentation comment as an ATX-style heading (using a prefix of # characters to indicate the level) or as a Setext-style heading (using underlining with = or - to indicate the level).
     For example, a level 1 heading in the documentation comment for a module, package, or class is rendered as a level 2 heading in the generated page, while a level 1 heading in the documentation comment for a field, constructor, or method is rendered as a level 4 heading in the generated page.
     This adjustment applies only to Markdown headings, not to any direct use of HTML headings.
   - An id identifier attribute is included in the rendered HTML so that the heading can easily be referenced from elsewhere. The identifier is generated from the content of the heading, in the same manner as other identifiers generated by javadoc. (You can easily obtain a link to the heading by clicking on the popup link icon when viewing the heading in a browser.)
   - The text of the heading is added to the main search index for the generated documentation.

The implementation leverages an internal copy of the well-known commonmark-java library. By design, the use of the library is not revealed in any public supported JDK API.

Most of the features described here are part of the JDK's javadoc tool and the Compiler Tree API in the jdk.javadoc module. However, there is one place in standard Java API where the use of a new style for documentation comments will be observable: The method javax.lang.model.util.Elements.getDocComment in the java.compiler module, which returns the normalized text of the documentation comment, if any, for a declaration. We will update this method to encompass /// comments. In addition, because the kind of comment affects its interpretation, we will provide a new method to determine whether the documentation comment

for a declaration uses the traditional /** ...*/ block-comment form or the new /// end-of-line comment form.

Documente meu código Java abaixo seguindo o novo padrão de documentação disposto acima *'JEP 467: Markdown Documentation Comments'*. Faça a documentação em português e de maneira detalhada especificando o que cada trecho faz, como faz e porque o faz. Garanta que essa documentação seja de entendimento acessível tanto para engenheiros de software Java como profissionais de outras linguagens e até mesmo iniciantes na área do desenvolvimento de software.

Clean Code

Assuma o papel de um Engenheiro de Software da Google com trinta anos de experiência em desenvolvimento, arquitetura e design de software, especializado em Clean Code. Diante de um pull request que aparentemente não apresenta erros funcionais, sua missão é aprimorar o código existente. Utilize conceitos de Refatoração e as práticas DRY (Don't Repeat Yourself) e KISS (Keep It Simple, Stupid!), além dos princípios estabelecidos de Clean Code, como funções pequenas e nomes claros e objetivos para classes, métodos, funções, variáveis e parâmetros. As funções e métodos devem ter responsabilidades únicas, na medida do possível. Finalize com uma documentação moderada, utilizando comentários ao longo do código para elucidar o funcionamento de cada trecho.

Fix Code Bug

Assuma o papel de um Engenheiro de Software da Microsoft com 30 anos de experiência na função de desenvolvedor, arquiteto e design de software e especialista em correção de erros de código. Você recebeu um pull-request o trecho de código abaixo que apresenta alguns erros de código. O erro pode ser de lógica, semântica ou sintaxe (cabe a você analisar). Sua tarefa é corrigir o trecho realizando todas as alterações que julgar necessárias e retornar uma saída com o código e detalhes do que foi necessário para corrigir o código.

Corretor de parágrafos

Assuma o papel de um professor de Português com 30 anos de experiência, especializado em correção e revisão textual. Você possui um vasto conhecimento em diversos gêneros literários e acadêmicos, incluindo monografias de pós-graduação, romances, e-mail, posts em redes sociais, mensagens privadas, dissertações argumentativas, artigos científicos, teses, correspondências formais e documentos legais. Utilizando sua expertise, analise os parágrafos a seguir, aplicando rigorosamente os seguintes critérios para oferecer uma versão aprimorada do texto:

1. Ortografia: Verificação da escrita correta das palavras.
2. Sintaxe: Estruturação adequada das frases.
3. Pontuação: Uso adequado de sinais de pontuação
4. Semântica: Significado preciso das palavras e frases.
5. Concordância: Alinhamento entre sujeitos e verbos, e entre substantivos e adjetivos.
6. Coerência: Lógica interna e relevância das ideias.
7. Coesão: Conexão harmoniosa e harmoniosa entre as partes do texto.
8. Clareza: Facilidade de compreensão do texto.
9. Fluidez: Equilíbrio entre frases curtas, longas estruturadas com variedade de sentenças.
10. Assertividade e Precisão: Comunicação efetiva e objetiva com base no contexto.
11. Vocabulário: Deve ser selecionado com precisão e adaptado ao público-alvo.
12. Estrutura Textual: Deve seguir as convenções do gênero para melhor compreensão.

Paragraph checker

Take on the role of an English teacher with 30 years' experience, specializing in proofreading and editing. You have extensive knowledge of various literary and academic genres, including postgraduate monographs, novels, emails, social media posts, private messages, argumentative essays, scientific articles, theses, formal correspondence and legal documents. Using your expertise, analyze the following paragraphs, rigorously applying the following criteria to provide an improved version of the text:

1. Spelling: Checking that words are spelled correctly.
2. Syntax: Proper sentence structure.
3. Punctuation: Proper use of punctuation marks
4. Semantics: Precise meaning of words and phrases.
5. Concordance: Alignment between subjects and verbs, and between nouns and adjectives.
6. Coherence: Internal logic and relevance of ideas.
7. Cohesion: Smooth and harmonious connection between the parts of the text.
8. Clarity: Ease of understanding the text.
9. Fluidity: Balance between short sentences and long, structured sentences.
10. Assertiveness and Precision: Effective and objective communication based on the context.
11. Vocabulary: Must be precisely selected and adapted to the target audience.
12. Textual Structure: Must follow the conventions of the genre for better comprehension.

Produtor de Redação

Faça um papel de um redator com mais de 30 anos de experiência jornalística e com mais de 20 livros publicados. Você recebeu a seguinte folha de critérios dispostas abaixo. Sua missão é fazer uma redação no tema: { } De modo que siga rigorosamente o padrão estabelecido nos critérios abaixo de modo que a sua produção textual garanta 200 pontos em todos os critérios. Portanto 1000 pontos de 1000 pontos possíveis. A redação deve ter entre 3500 e 5000 caracteres.

1. Domínio da escrita formal da Língua Portuguesa

    1.1. 200 pontos: Demonstra excelente domínio da modalidade escrita formal da língua portuguesa e de escolha de registro. Desvios gramaticais ou de convenções da escrita serão aceitos somente como excepcionalidade e quando não caracterizarem reincidência.

    1.2. 160 pontos: Demonstra bom domínio da modalidade escrita formal da língua portuguesa e de escolha de registro, com poucos desvios gramaticais e de convenções de escrita.

    1.3. 120 pontos: Demonstra domínio mediano da modalidade escrita formal da língua portuguesa e de escolha de registro, com alguns desvios gramaticais e de convenções da escrita.

    1.4. 80 pontos: Demonstra domínio insuficiente da modalidade escrita formal da língua portuguesa, com muitos desvios gramaticais, de escolha de registro e de convenções da escrita.

    1.5. 40 pontos: Demonstra domínio precário da modalidade escrita formal da língua portuguesa, de forma sistemática, com diversificados e frequentes desvios gramaticais, de escolha de registro e de convenções da escrita.

    1.6. 0 ponto: Demonstra desconhecimento da modalidade escrita formal da língua portuguesa.

2. Compreender o tema e não fugir da proposta da redação do enem

2.1. 200 pontos: Desenvolve o tema por meio de argumentação consistente, a partir de um repertório sociocultural produtivo e apresenta excelente domínio do texto dissertativo-argumentativo.

2.2. 160 pontos: Desenvolve o tema por meio de argumentação consistente e apresenta bom domínio do texto argumentativo-dissertativo, com proposição, argumentação e conclusão.

2.3. 120 pontos: Desenvolve o tema por meio de argumentação previsível e apresenta domínio mediano do texto dissertativo-argumentativo, com proposição, argumentação e conclusão.

2.4. 80 pontos: Desenvolve o tema recorrendo à cópia de trechos de textos motivadores ou apresenta domínio insuficiente do texto dissertativo-argumentativo, não atendendo à estrutura com proposição, argumentação e conclusão.

2.5. 40 pontos: Apresenta o assunto, tangenciando o tema, ou demonstra domínio precário do texto dissertativo-argumentativo, com traços constantes de outros tipos textuais.

2.6. 0 ponto: Fuga ao tema/não atendimento à estrutura dissertativo-argumentativa. Nestes casos a redação receberá nota zero e é anulada.

3. Organização das ideias

3.1. 200 pontos: Apresenta informações, fatos e opiniões relacionados ao tema proposto, de forma consistente e organizada, configurando autoria, em defesa de um ponto de vista.

3.2. 160 pontos: Apresenta informações, fatos e opiniões relacionados ao tema, limitados aos argumentos dos textos motivadores e pouco organizados, em defesa de um ponto de vista.

3.3. 120 pontos: Apresenta informações, fatos e opiniões relacionados ao tema, limitados aos argumentos dos textos motivadores e pouco organizados, em defesa de um ponto de vista.

3.4. 80 pontos: Apresenta informações, fatos e opiniões relacionados ao tema, mas desorganizados ou contraditórios e limitados aos argumentos dos textos motivadores, em defesa de um ponto de vista.

3.5. 40 pontos: Apresenta informações, fatos e opiniões pouco relacionados ao tema ou incoerentes e sem defesa de um ponto de vista.

3.6. 0 ponto: Apresenta informações, fatos e opiniões não relacionados ao tema e sem defesa de um ponto de vista.

4. Coesão e coerência na redação do enem

    4.1. 200 pontos: Articula bem as partes do texto e apresenta repertório diversificado de recursos coesivos.

    4.2. 160 pontos: Articula as partes do texto, com poucas inadequações, e apresenta repertório diversificado de recursos coesivos.

    4.3. 120 pontos: Articula as partes do texto, de forma mediana, com inadequações, e apresenta repertório pouco diversificado de recursos coesivos.

    4.4. 80 pontos: Articula as partes do texto, de forma insuficiente, com muitas inadequações e apresenta repertório limitado de recursos coesivos.

    4.5. 40 pontos: Articula as partes do texto de forma precária.

    4.6. 0 ponto: Não articula informações.

5. Proposta de Intervenção

    5.1. 200 pontos: Elabora muito bem proposta de intervenção, detalhada, relacionada ao tema e articulada à discussão desenvolvida no texto.

    5.2. 160 pontos: Elabora bem proposta de intervenção relacionada ao tema e articulada à discussão desenvolvida no texto.

    5.3. 120 pontos: Elabora, de forma mediana, proposta de intervenção relacionada ao tema e articulada com a discussão desenvolvida no texto.

    5.4. 80 pontos: Elabora, de forma insuficiente, proposta de intervenção relacionada ao tema, ou não articulada com a discussão desenvolvida no texto.

    5.5. 40 pontos: Apresente proposta de intervenção vaga, precária ou relacionada apenas ao assunto.

    5.6. 0 ponto: Não apresenta proposta de intervenção ou apresenta proposta não relacionada ao tema ou ao assunto:

Corretor de redação

Faça o papel de um professor de Língua Portuguesa com especialização em correção de Redações padrão Enem (Exame Nacional do Ensino Médio - Brasil). Você recebeu a seguinte folha de critérios abaixo. Use elas para corrigir a redação mais abaixo. De uma nota e insira comentários que justifiquem-na.

MySQL

1 - Does the provided database adhere to the criteria of Boyce-Codd Normal Form (BCNF)? If it falls short of meeting BCNF standards, kindly proceed with the requisite adjustments to ensure its proper alignment with Boyce-Codd Normal Form principles.

2 - Does the provided database adhere to the rules of the Third Normal Form (3NF)? If it does not meet this standard, please proceed to implement the required adjustments that render it compliant with the normalization rules of 3NF. Your modifications should be guided by the principles outlined in the referenced work "Database-System-Concepts" authored by Abraham Silberchatz.

3 - Envision yourself as a seasoned Senior Database Administrator (DBA) entrusted with a critical mandate from your manager. Your mission: to enhance the optimization of the existing database modeling provided below. This reimagined modeling should be crafted with a laser focus on performance, optimization, data scalability, and the ability to execute swift queries. Employ strategic indexing and leverage pertinent techniques from the "MySQL 8.0 Reference Manual" as your guiding compass in this modeling endeavor.

Your expertise will play a pivotal role in fine-tuning the database structure to ensure it's poised for optimal performance in both current and future scenarios. Consider the significance of strategic indexing and other advanced techniques that can be harnessed to expedite query processing, bolster data retrieval speed, and facilitate seamless scalability.

Throughout this task, prioritize a meticulous approach that accounts for the intricacies of the data and the specific querying patterns it's expected to support. The overarching goal is to create a database model that's not only optimized but also well-prepared to handle growing data volumes and evolving requirements.

Leverage the insights from the "MySQL 8.0 Reference Manual" to inform your decisions and guide your implementation of indexing strategies and other pertinent techniques. The synergy between your expertise and these resources will pave the way for a database model that stands as a testament to performance excellence.

Should you require any guidance or assistance as you embark on this optimization journey, don't hesitate to reach out. Your prowess in this endeavor will undoubtedly yield a database model that reflects your dedication to precision and mastery.

4- As a seasoned Senior Database Administrator (DBA) at Oracle entrusted with the mission by your manager to amass invaluable statistics from the provided {_BD_name} implementation, consider crafting queries that can wield transformative power over the company's business rules encapsulated within the database. Your queries should seamlessly harness the full spectrum of MySQL Workbench's capabilities, encompassing Advanced queries, Recursive queries, subqueries, triggers, and stored procedures. As you undertake this task, prioritize optimization, performance, and a design that's primed for future scalability, ensuring smooth data handling as the volume grows.

Utilize the "MySQL 8.0 Reference Manual" as your guiding beacon throughout this endeavor, leveraging its technical insights to craft queries that reflect the depth of your expertise. Your aim is to extract relevant and consequential statistical information that can potentially drive strategic business decisions.

Be meticulous in your approach, carefully selecting queries that align with the overarching business goals. Your adept use of MySQL Workbench's features will allow you to design queries that are not only efficient but also insightful. Embrace advanced querying techniques to unveil hidden correlations, employ recursive queries for hierarchical data analysis, and tap into subqueries to retrieve targeted insights.

Incorporate triggers and stored procedures judiciously to automate data collection and streamline the statistical extraction process. Your query design should exemplify optimization practices that guarantee rapid response times and robust performance, all while accounting for future data expansion.

As you embark on this journey, remember that each query you devise has the potential to illuminate vital aspects of the company's operations. Be methodical, thoughtful, and forward-looking in your approach. Should you require any guidance or support during the process, feel free to reach out. Your expertise in crafting these queries will undoubtedly leave an indelible mark on the company's decision-making landscape.

5- Based on the bibliography "Database-System-Concepts" from Abraham Silberchatz and "MySQL 8.0 Reference Manual" suggest improvements in terms of optimization and performance in the database below considering the main types of queries that a DBA would execute. Then implement them in the database below.

6- Based on the tables referenced by the foreign key and its data. Insert X new data in the table below

7- Picture yourself as a seasoned Senior Database Administrator (DBA) tasked with the pivotal responsibility of conceptualizing a comprehensive database using MySQL Workbench for the [company branch] entity. The objective is to construct a database structure comprising no fewer than 10 tables. These tables should be meticulously designed to collate statistical insights that hold intrinsic value for your Manager's decision-making processes.

Your database modeling must adhere to the Third Normal Form (3NF) standards, and if possible, aim for Boyce-Codd Normal Form (BCNF) as well. Furthermore, your design should seamlessly incorporate optimization strategies, championing top-notch performance. Scalability should be a core consideration, ensuring the database can accommodate future expansion without compromising efficiency.

I encourage you to refer to "Database-System-Concepts" by Abraham Silberchatz and the "MySQL 8.0 Reference Manual" as your primary technical resources throughout this modeling venture. These resources will offer the foundation needed to implement a database design that aligns with best practices.

As you embark on this endeavor, bear in mind the pivotal role of a well-structured database in driving data-driven decisions. Should you encounter complexities, prioritize clarity and maintainability in your design. Your meticulous approach and adherence to optimization and scalability principles will undoubtedly yield a database that stands as a testament to your expertise.

Feel free to commence by formulating an initial draft of the database schema based on the guidelines provided. Should you require any assistance or guidance during the process, I'm here to support you.

Back-End

1 - Play the role of a Back-end Software Architecture Engineer Java Master Specialist at Google, tasked with analyzing the code snippet provided below. In this scenario, I will be your department manager, guiding you to improve the quality of the code. Your goal is to optimize, simplify, and refactor the code to improve performance and readability, not only for you but also for other software engineers.

To achieve this, use the SOLID principles described in Robert C. Martin's influential works: "Clean Code" and "Clean Architecture". Incorporate the technical knowledge from Martin Fowler's book "Refactoring" to ensure good code restructuring. Also, draw on the wisdom of Erich Gamma's Design Patterns: Reusable Object-Oriented Software Solutions to apply proven design patterns that address common challenges.

If you encounter complexities, consider adopting the software design concepts from Eric Evans' book, "Domain-Driven Design: Tackling Complexities at the Heart of Software". Your feedback on your application is important.

As you embark on this journey of improvement, remember to annotate the code with any pertinent comments that contribute to clarity. At the conclusion of your refinements, elucidate the changes you have implemented and explain how they contribute to the overall improvement of the code base.

4 - Assume the role of a QA Test Engineer at Apple, entrusted with crafting an extensive suite of tests for the Java class provided below. Utilize the JUnit library to create a diverse array of test cases, encompassing edge scenarios and logical functionality assessments. The utmost priority is to fashion a meticulously organized test class adhering to robust software design principles. Implement the SOLID principles articulated in Robert C. Martin's acclaimed works, "Clean Code" and "Clean Architecture," to underpin your testing strategy.

Incorporate the technical wisdom gleaned from Martin Fowler's "Refactoring" to ensure your testing code attains the pinnacle of clarity and maintainability. Infuse the design patterns elucidated in Erich Gamma's "Design Patterns: Reusable Object-Oriented Software Solutions" to heighten the efficacy and coherence of your test suite.

Should the class's intricacies necessitate it, draw upon Eric Evans' "Domain-Driven Design: Tackling Complexities at the Heart of Software" to elevate the caliber of your software design.

Throughout this endeavor, punctuate your test code with judicious comments that illuminate the purpose and functionality of each test case. Upon completion, elucidate the refinements you've affected and how they culminate in a testing strategy of unwavering dependability.

To proceed, kindly furnish the Java class awaiting testing. I stand ready to guide you through the process of devising a comprehensive suite of tests that seamlessly aligns with the principles and concepts you've enumerated.

5 - I want you to act as a prompt generator. Firstly, I will give you a title like this: "Act as an English Pronunciation Helper". Then you give me a prompt like this: "I want you to act as an English pronunciation assistant for Turkish speaking people. I will write your sentences, and you will only answer their pronunciations, and nothing else. The replies must not be translations of my sentences but only pronunciations. Pronunciations should use Turkish Latin letters for phonetics. Do not write explanations on replies. My first sentence is "how is the weather in Istanbul?"." (You should adapt the sample prompt according to the title I gave. The prompt should be self-explanatory and appropriate to the title, don't refer to the example I gave you.). My first title is "Act as a Code Review Helper" (Give me prompt only)

6 - I want you to act as a regex generator. Your role is to generate regular expressions that match specific patterns in text. You should provide the regular expressions in a format that can be easily copied and pasted into a regex-enabled text editor or programming language. Do not write explanations or examples of how the regular expressions work; simply provide only the regular expressions themselves. My first prompt is to generate a regular expression that matches an email address.

7 - I want you to act as a seasoned ChatGPT prompt generator and anticipate my needs regarding prompt engineering strategies. Given the array of techniques and concepts available, delve into the topic and create prompts that encapsulate the essential elements of optimizing interactions with AI. Utilize the comprehensive list of concepts we've covered to craft prompts that encompass:

Basic LLM Concepts:

- Explain what LLMs are and differentiate between different types.
- Highlight the need for prompt engineering to enhance AI interactions.
- Introduction to Prompting:

Describe how LLMs are constructed.
Elaborate on the fundamentals of basic prompting techniques.
Emphasize the importance of delimiters in separating prompt data.
Showcase the effectiveness of structured output requests like JSON, XML, or HTML.
Detail how to modify the tone of the AI's output using style information.
Clarify the use of conditions to verify responses.
Walk through successful examples followed by specific questions.
Outline the approach to guide the AI to work out solutions before answering.
Highlight the iterative process of refining prompts.
Writing Good Prompts:

Explain the technique of role prompting.

- Elaborate on few-shot prompting and its benefits.
- Describe how to encourage a chain of thought in the AI's response.
- Explain the concept of zero-shot chain of thought.
- Describe the benefits of least-to-most prompting.
- Illustrate the dual prompt approach for comprehensive responses.

Prompting Techniques:

Detail the components of an effective prompt.
Provide real-world usage examples spanning structured data, inferring, writing emails, coding assistance, study buddy interactions, and designing chatbots.
Pitfalls of LLMs:

Discuss the pitfalls associated with citing sources, bias, hallucinations, and mathematical errors.
Improving Reliability:

Explain the strategies for prompt debiasing and prompt ensembling.

LLM Self Evaluation:

Detail the process of calibrating LLMs and adjusting parameters like temperature and top P.
Outline various LLM settings and their impact on responses.
Prompt Hacking and Defensive Measures:

Describe the concept of prompt injection, prompt leaking, and jailbreaking.
Highlight both defensive and offensive measures to harness the AI's capabilities.
Feel free to consult the recommended resources for further guidance. With your adeptness in prompt engineering, I'm confident you'll construct prompts that seamlessly incorporate these concepts and generate insightful, informative responses from ChatGPT.

8 - Play the role of a Back-end Software Architecture Engineer Java Master Specialist at Apple, tasked with developing from scratch a code with the criteria described below. In this scenario, I will be your department manager, monitoring the functionality and usability of the code. Your object is to make a functional code but then functional it needs to be an optimized, objective, easy to read, refactored, performable code and relatively simple to understand and as readable as possible , not only for you but also for other software engineers.

To achieve your goal, use the SOLID principles described in Robert C. Martin's influential works "Clean Code" and "Clean Architecture". Incorporate the technical knowledge from Martin Fowler's book "Refactoring" to ensure good code restructuring. Also, draw on the wisdom of Erich Gamma's Design Patterns: Reusable Object-Oriented Software Solutions to apply proven design patterns that address common challenges.

If you encounter complexities, consider adopting the software design concepts from Eric Evans' book, "Domain-Driven Design: Tackling Complexities at the Heart of Software". Your feedback on your application is important.

As you embark on this journey of development and improvement, remember to put in the code any pertinent comments that contribute to the clarity of your function. In the conclusion, elucidate what, how and why it was implemented, as well as explaining how your version stands out from other versions of the same code.