

StringCheck プログラム 詳細設計書

短文投稿を行うCLIプログラムの実装仕様と、言語非依存の処理フロー

実装対象ファイル名

StringCheck.java

前提環境

Java 11 以降

作成日

2026-09-07

目次

- | | |
|-----------------|------------------|
| 1. 概要 | 6. データ項目定義 |
| 2. 前提知識 | 7. ログ出力仕様 |
| 3. 全体構成 | 8. 画面入出力例 |
| 4. 処理仕様（メソッド詳細） | 9. フローチャート |
| 5. 入力チェック仕様 | 10. 疑似コード（言語非依存） |
| | 11. テストケース一覧 |

1. 概要

本書は、短文投稿を行うCLIプログラム『StringCheck』の詳細設計書です。研修生はこの設計書をもとに、まっさらな状態からプログラムを実装してください。

1.1 プログラム概要

ユーザーがキーボードから入力した短い文章（投稿文）を受け取り、「投稿されました」として画面に表示する、シンプルなCLIアプリケーションです。ただし、入力された内容によっては投稿を受け付けず、再入力を求める必要があります。

1.2 実装仕様（必須要件）

プログラムは、次の a~d の仕様をすべて満たすこと。

記号	仕様内容
a	何も入力していなかった場合は、「何も入力されていません」と表示し、再度「いまなにしてる？」を表示して入力待ち状態に戻すこと。
b	141文字以上の場合は、「投稿文が長すぎます」と表示し、再度入力待ち状態に戻すこと。
c	スペースのみで入力されている場合は、「何も入力されていません」と表示し、再度入力待ち状態に戻すこと。
d	入力文字列の両端にスペースがあった場合は、そのスペースを除去すること。

上記いずれの条件にも該当しない場合は、入力された文章（両端の空白を除去したもの）を投稿稿として画面に表示すること。

補足： c は、d（両端の空白除去）を行った結果として文字列が空になるケースを含みます。a・b・c・d をどの順番で判定するかは「5. 入力チェック仕様」で規定します。

2. 前提知識

本課題の実装には、Java 標準クラスライブラリの API を利用します。以下に挙げる機能について、**Java SE API ドキュメント**を自分で参照し、使い方を調べたうえで実装してください。ここでは「何をする必要があるか」のみを示し、使用するクラス名やメソッドはあえて記載していません。

- 標準入力から1行分の文字列を読み取る方法
- 文字列の前後にある空白文字（半角・全角を含む）を除去する方法
- 文字列の長さ（文字数）を取得する方法
- プログラムの実行記録をファイルへ出力する仕組み（ログ出力）※基本的な使い方は「2.1 java.util.Logger の基本」を参照

参考：Java SE 21 API 仕様（日本語版）(docs.oracle.com/javase/jp/21/docs/api/index.html、実装環境に合わせて該当バージョンを選択してください)。「Java + 実現したいこと」で検索すると、目的のクラス・メソッドにたどり着きやすくなります。自分で見つけたクラスやメソッドの仕様（引数・戻り値・挙動）は、必ず公式ドキュメントで裏を取ってから使用すること。

2.1 java.util.Logger の基本

ログ出力の仕組み（java.util.logging パッケージ）は初めて扱う人も多いため、基本的な考え方だけここで解説します。ただし、これはあくまで一般的な使い方の説明です。本課題で実際に必要な設定値やメッセージは「4.3 setUpLogger()」「7. ログ出力仕様」で規定するので、そちらを満たすように自分でコードを組み立てて実装してください。

java.util.logging パッケージは、次の4つの部品が連携して動作します。

部品	役割
Logger	ログを出力する際の入り口となるオブジェクト。 <code>logger.info(...)</code> のようにメッセージを渡す。
Level	ログの重要度を表す区分（下記参照）。Loggerやハンドラに設定することで、どこまで細かいログを扱うかを制御する。
Handler	ログの出力先を制御する部品。コンソールに出す <code>ConsoleHandler</code> や、ファイルに出す <code>FileHandler</code> などがある。Loggerに複数のHandlerを追加できる。
Formatter	ログ1件を実際にどんな文字列（見た目）に整形するかを決める部品。簡潔な1行形式にする <code>SimpleFormatter</code> などがある。Handlerに対して設定する。

1. Loggerの取得とログレベルの設定

Loggerは通常、クラスごとに1つ、次のように取得します。

```
Logger logger = Logger.getLogger(何らかのクラス.class.getName());
```

ログレベルは、重要度の高い順に次のように定義されています。

```
SEVERE > WARNING > INFO > CONFIG > FINE > FINER > FINEST
```

加えて、「すべて出力する」ことを意味する `Level.ALL` と、「何も出力しない」ことを意味する `Level.OFF` があります。`logger.setLevel(Level.ALL)` のようにすると、そのLoggerが扱うログの範囲を指定できます。

2. ログメッセージの出力

主なログレベルには専用のメソッドが用意されています。

```
logger.info("通常の経過を記録するメッセージ");
logger.warning("警告レベルのメッセージ");
logger.severe("重大なエラーを記録するメッセージ");
```

専用メソッドがないレベルは、`logger.log(Level.CONFIG, "メッセージ")` のように汎用の `log` メソッドでレベルを指定して出力できます。

3. Handler（出力先）の追加

Loggerを作っただけでは、どこにもログは出力されません。Loggerに1つ以上のHandlerを追加することで、初めて出力先が決まります。

```
Logger.addHandler(何らかのHandlerのインスタンス);
```

ファイルへ出力する `FileHandler` は、次のような形のコンストラクタを持ちます（実際に使用する値は課題ごとに異なります）。

```
new FileHandler(String パターン, int limit, int count, boolean append)
```

- **パターン**：出力するファイル名。`%g` と書いた部分が世代番号（0, 1, 2...）に置き換わる。
- **limit**：1つのファイルに書き込む最大バイト数のおおよその目安。これを超えると次の世代のファイルに切り替わる。
- **count**：保持しておく世代数。
- **append**：既存のファイルがある場合に、追記するかどうか。

`FileHandler`の生成は入出力エラー（`IOException`）が発生する可能性があるため、try-catchで例外を処理する必要があります。

4. Formatter（出力形式）の設定

Handlerには、ログの見た目を整形するFormatterを設定できます。

```
何らかのHandlerのインスタンス.setFormatter(何らかのFormatterのインスタンス);
```

標準で用意されている `SimpleFormatter` は、日時・呼び出し元・メッセージなどを簡潔な1行程度の形式にまとめてくれます。

組み合わせた例（本課題とは別の、一般的な例）

```
Logger sampleLogger = Logger.getLogger("SampleApp");

try {
    FileHandler handler = new FileHandler("sample_%g.log", 5000, 3, false);
    handler.setFormatter(new SimpleFormatter());
    sampleLogger.addHandler(handler);
    sampleLogger.setLevel(Level.ALL);
} catch (IOException e) {
    sampleLogger.log(Level.WARNING, "ログファイルの初期化に失敗しました", e);
}

sampleLogger.info("アプリケーションを起動しました");
sampleLogger.warning("設定ファイルが見つかりません。デフォルト値を使用します");
```

上記はあくまで `java.util.logging` の一般的な使い方を示すサンプルです。ファイル名や `limit・count` の数値、追記モードの真偽値はこの例のためのものであり、本課題の要件ではありません。本課題で実際に必要な設定値は「4.3 `setUpLogger()`」の表を、出力すべきメッセージ・タイミング・レベルは「7. ログ出力仕様」を確認し、それらを満たすように自分でコードを組み立てて実装してください。

3. 全体構成

次のクラス構成で実装すること。

種別	名前	役割
フィールド	<code>logger</code> (static final)	クラス全体で共有するログ出力用オブジェクト。
メソッド	<code>main(String[] args)</code>	プログラムの開始点。入力受付ループと結果表示を行う中心処理。
メソッド	<code>inputString()</code>	標準入力から1行読み取り、両端の空白を除去して返す。
メソッド	<code>setUpLogger()</code>	ログをファイルへ出力するための準備（設定）を行う。

`main` は起動時に `setUpLogger` を1回呼び出し、その後の入力受付ループの中で毎回 `inputString` を呼び出す構成とすること。

4. 処理仕様（メソッド詳細）

4.1 `main(String[] args)`

役割： プログラム全体の流れを制御すること。ログ出力の準備、入力受付ループ（未入力・文字数チェック）、結果の表示・記録までを行うこと。

1. `setUpLogger()` を呼び出し、ログ出力の準備を行うこと。
2. ログレベルを ALL（すべて記録する）に設定すること。
3. 起動を知らせるログ（INFO）を出力すること。
4. 変数 `str` を用意すること。
5. 次の手順を、正しい入力が見られるまで繰り返すこと。
 1. 画面に「いまなにしてる？」と表示すること。
 2. `inputString()` を呼び出し、結果を `str` に受け取ること。
 3. `str` の文字数が0であれば、「何も入力されていません」を表示し、ログ（SEVERE）に記録して、手順(a)へ戻ること。

4. str の文字数が1～140文字の範囲に収まっていなければ（＝141文字以上であれば）、「投稿文が長すぎます」を表示し、ログ（SEVERE）に記録して、手順(a)へ戻ること。
5. どちらの条件にも当てはまらなければ、繰り返しを終了すること。
6. 空行を1行出力すること。
7. 「↓↓↓↓↓↓ 投稿されました ↓↓↓↓↓↓」を表示すること。
8. str の内容を表示すること。
9. 投稿内容をログ（INFO）に記録すること。
10. 終了を知らせるログ（INFO）を出力し、処理を終えること。

4.2 inputString()

役割： 標準入力から1行分の文字列を読み取り、両端の空白を除去して呼び出し元に返すこと。

- **引数：** なし
- **戻り値：** 前後の空白（半角・全角を含む）を除去した文字列

両端の空白除去（仕様d）は、このメソッドの中で完結させ、main 側では文字数の判定のみを行う構成にすること。実現に使うAPIは「2. 前提知識」を参考に自分で調べること。

4.3 setUpLogger()

役割： ログをファイルに出力するための準備を行い、共有の logger に設定を反映すること。

設定項目	設定値
出力ファイル名パターン	StringCheck_%g.log （%g は世代番号 0,1,2… に置き換わる）
ファイル切替サイズ	100 バイト
世代数	10
追記モード	true
出力形式	人が読みやすい簡潔な1行形式
出力レベル	ALL

異常時の挙動： ログ出力の準備に失敗した場合（例：ファイルが作成できない環境など）は、例外を捕まえて「ログ出力機能が正常に動作していません。」という内容を SEVERE レベルでログに記録すること。この場合でも、プログラム本体（投稿処理）は継続して動作すること。

ログ出力の実現には、Java標準のロギング機能（java.util.logging パッケージ）を利用します。具体的なクラスや設定方法はAPIドキュメントを参照して自分で調べること。

5. 入力チェック仕様

「1.2 実装仕様」の a～d を、判定条件として整理する。

仕様	チェック内容	条件（言語非依存表現）	表示メッセージ	ログ
a	未入力チェック	<code>str</code> の文字数が 0	「何も入力されていません」	SEVERE
b	文字数上限チェック	<code>str</code> の文字数が 1～140 の範囲外（＝141文字以上）	「投稿文が長すぎます」	SEVERE
c	空白のみ入力チェック	入力全体が空白文字のみで構成されている	「何も入力されていません」	SEVERE
d	両端空白の除去	—（判定ではなく整形処理）	—	—

判定順序：まず a（文字数0か）を判定し、そのあとで b（1～140文字の範囲内か）を判定すること。この順序により、bの判定に入った時点で文字数は必ず1文字以上あることが保証される。また、d（両端空白除去）を先に行うことで、c は自然に a の判定に含まれる形になる。

6. データ項目定義

項目名	型	説明	取りうる範囲・制約
<code>str</code>	文字列	投稿される本文。両端の空白を除去済み。	正常時：1～140文字／異常時：0文字 または 141文字以上（いずれも再入力対象）

7. ログ出力仕様

タイミング	レベル	出力内容
プログラム起動時	INFO	「StringCheck が起動しました。」
未入力（または空白のみ）と判定した時	SEVERE	「何も入力されていません」
141文字以上と判定した時	SEVERE	「投稿文が長すぎます」＋入力内容
投稿確定時	INFO	「投稿された内容：」＋投稿内容
プログラム終了時	INFO	「StringCheck が終了しました。」
ログ出力の準備に失敗した時	SEVERE	「ログ出力機能が正常に動作していません。」

ログファイルの出力先や世代管理の設定は「4.3 `setUpLogger()`」の表を参照してください。

8. 画面入出力例

8.1 正常系（1回で投稿が成功する場合）

いまなにしてる？
ラーメンを食べています

↓↓↓↓↓↓ 投稿されました ↓↓↓↓↓↓
ラーメンを食べています

8.2 未入力の場合

いまなにしてる？

何も入力されていません
いまなにしてる？
（再入力を待つ）

8.3 スペースのみ入力した場合

いまなにしてる？

何も入力されていません
いまなにしてる？
（再入力を待つ）

8.4 141文字以上入力した場合

いまなにしてる？
（141文字以上の文章）
投稿文が長すぎます
いまなにしてる？
（再入力を待つ）

8.5 両端に空白がある入力の場合

いまなにしてる？
今日は晴れ

↓↓↓↓↓↓ 投稿されました ↓↓↓↓↓↓
今日は晴れ

両端の全角スペースが除去され、「今日は晴れ」のみが投稿される。

9. フローチャート

どのプログラミング言語でも実装できるように、Java固有の記法（正規表現など）を使わず、「文字数」「はい／いいえ」といった一般的な言葉で条件を表しています。

9.1 メイン処理 (main)

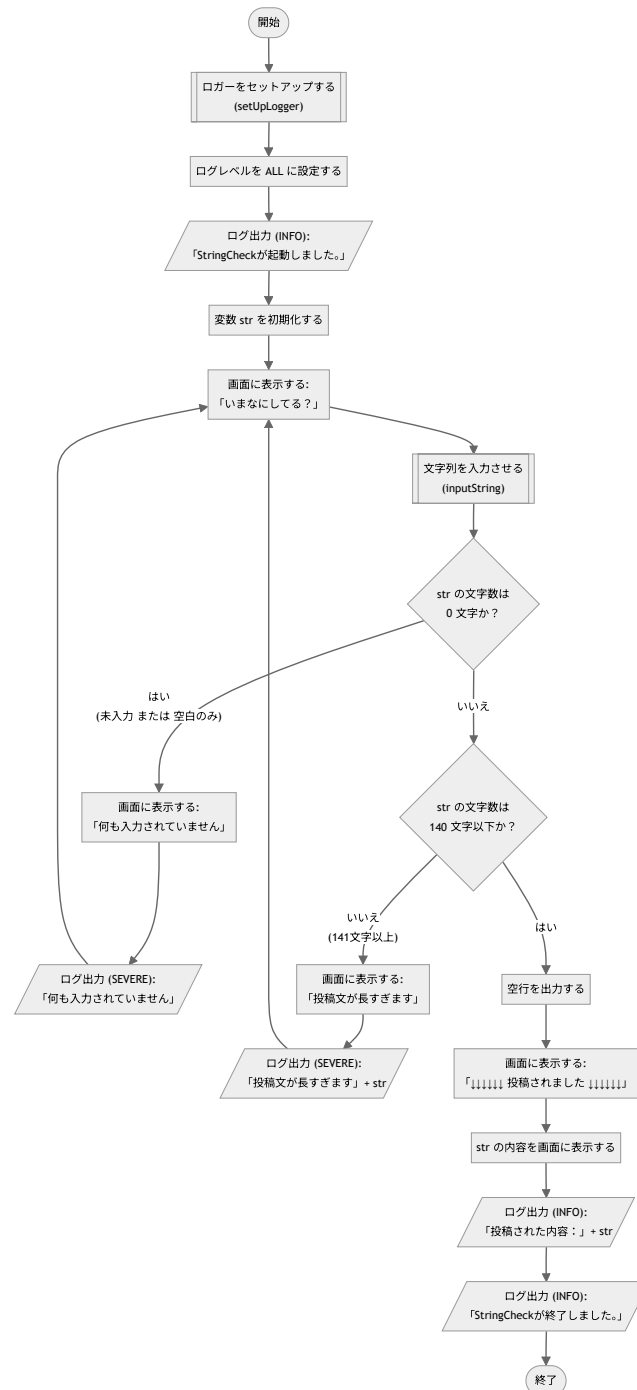
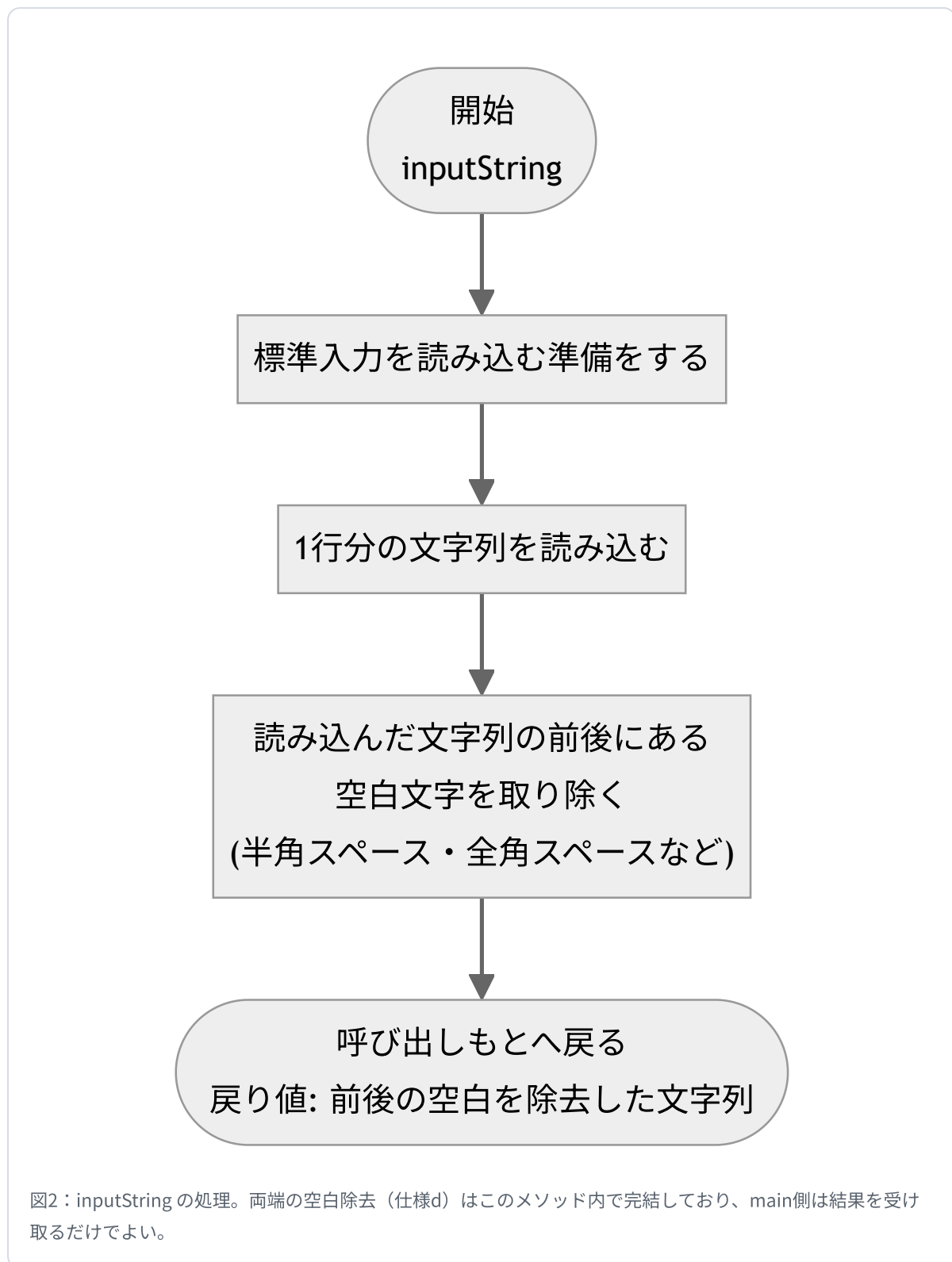


図1：main のメイン処理。未入力（文字数0）と文字数超過（141文字以上）のどちらの場合も、メッセージ表示とログ記録の後、入力受付（「いまなにしてる？」の表示）へ戻る。

9.2 文字列入力処理 (inputString)



9.3 ロガー初期化処理 (setUpLogger)

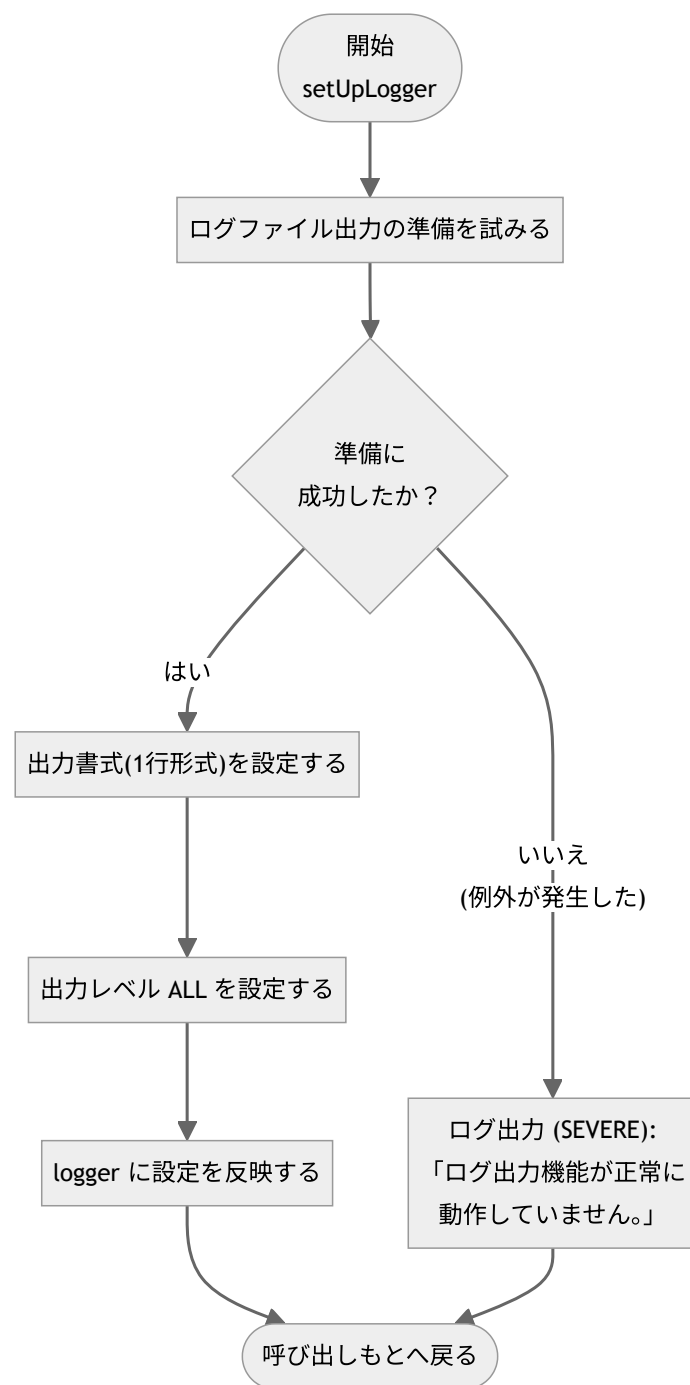


図3：setUpLogger の処理。準備に失敗しても例外を握りつぶし、プログラム本体は継続できるようにする。

10. 疑似コード（言語非依存）

フローチャートと対応する疑似コードです。特定の言語の構文には依存しない書き方にしています。

手続き main:

```
setUpLogger を呼び出す  
ログレベル を ALL に設定する  
ログ出力(INFO, "StringCheckが起動しました。")
```

```
str ← ""
```

繰り返す:

```
画面に "いまなにしてる?" を表示する  
str ← inputString を呼び出した結果
```

もし (strの文字数 = 0) ならば:

```
画面に "何も入力されていません" を表示する  
ログ出力(SEVERE, "何も入力されていません")  
この回の繰り返しを次へ進める (先頭に戻る)
```

もし (strの文字数 < 1 または strの文字数 > 140) ならば:

```
画面に "投稿文が長すぎます" を表示する  
ログ出力(SEVERE, "投稿文が長すぎます" + str)  
この回の繰り返しを次へ進める (先頭に戻る)
```

繰り返しを終了する

```
画面に空行を表示する  
画面に "↓↓↓↓↓↓ 投稿されました ↓↓↓↓↓↓" を表示する  
画面に str を表示する  
ログ出力(INFO, "投稿された内容:" + str)  
ログ出力(INFO, "StringCheckが終了しました。")
```

手続き終了

関数 inputString → 文字列を返す:

```
line ← 標準入力から1行読み取る  
結果 ← line の前後にある空白文字 (半角・全角) を除去したもの  
結果 を返す
```

関数終了

手続き setUpLogger:

試みる:

```
ファイル出力の準備を行う (
    ファイル名パターン: "StringCheck_%g.log",
    切り替えサイズ: 100バイト,
    世代数: 10,
    追記モード: true
)
```

```
出力書式 ← シンプルな1行形式  
出力レベル ← ALL  
Logger に設定を反映する
```

例外が発生した場合:

```
ログ出力(SEVERE, "ログ出力機能が正常に動作していません。")
```

手続き終了

11. テストケース一覧

実装後は、少なくとも次のケースで動作確認を行うこと。

No	入力内容	期待される画面表示	対応仕様
1	「ラーメンを食べています」(1~140文字の通常の文章)	投稿完了メッセージとともに投稿内容が表示される	— (正常系)
2	何も入力せずEnterのみ	「何も入力されていません」→再度入力待ち	a
3	半角スペースのみ (例: " ")	「何も入力されていません」→再度入力待ち	c (d経由でaに合流)
4	全角スペースのみ (例: " ")	「何も入力されていません」→再度入力待ち	c (d経由でaに合流)
5	ちょうど140文字の文章	投稿が成功する (境界値・正常系)	b (境界)
6	141文字の文章	「投稿文が長すぎます」→再度入力待ち	b (境界)
7	大幅に140文字を超える文章	「投稿文が長すぎます」→再度入力待ち	b
8	両端に半角・全角スペースが付いた通常の文章 (例: "今日は晴れ")	両端の空白が除去され「今日は晴れ」として投稿される	d
9	不正入力 (未入力・長すぎ) を数回続けた後、正しい文章を入力	都度エラーメッセージが表示された後、最終的に投稿が成功する	a, b (再入力ループの確認)