



GitHub Copilot

レガシーモダナイゼーション

Code to Doc / Doc to Code

ワークシヨップ

How do you use this tool, you ask? Follow me!





GitHub Copilot

Code to Doc / Doc to Code
ワークショップ

Who am I ?



Shinya Yanagihara

shinyay

```
{ "position": "Developer Global Black Belt", "location": "Tokyo,JP", "favorites": ["Spring", "Kotlin", "Developer Productivity"] }
```

Edit profile

24 followers · 12 following

GitHub **STAFF**

Tokyo

19:23 (UTC +09:00)

shinyay@github.com

<https://www.microsoft.com>

@yanashin18618

@yanashin@mastodon.social

in/shinyay

shinyayREADME,ed /



Java truly is an OOP language... As in:
OOPs I used Java!

Shinya Yanagihara's GitHub Stats

★ Total Stars Earned:	43
🕒 Total Commits (2024):	2.6k
🔗 Total PRs:	104
📄 Total Issues:	80
📅 Contributed to (last year):	1

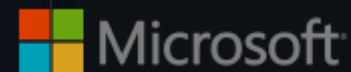


Most Used Languages



views 6469 Twitter Profile

Career



SNS

@shinyay

@yanashin18618

/in/shinyay

本日のアジェンダ

- GitHub Copilot 概要
- [WS1] Copilot Chat 利用
- [WS2] レガシーコードの分析
- [WS3] レガシーコード変換のための調査
- [WS4] レガシーコード設計書作成
- [WS5] Java クラス設計
- [WS6] Java コード開発
- [WS7] 実践: レガシー Java からの
モダナイゼーション
- ※適宜 休憩 & Ask the Expert

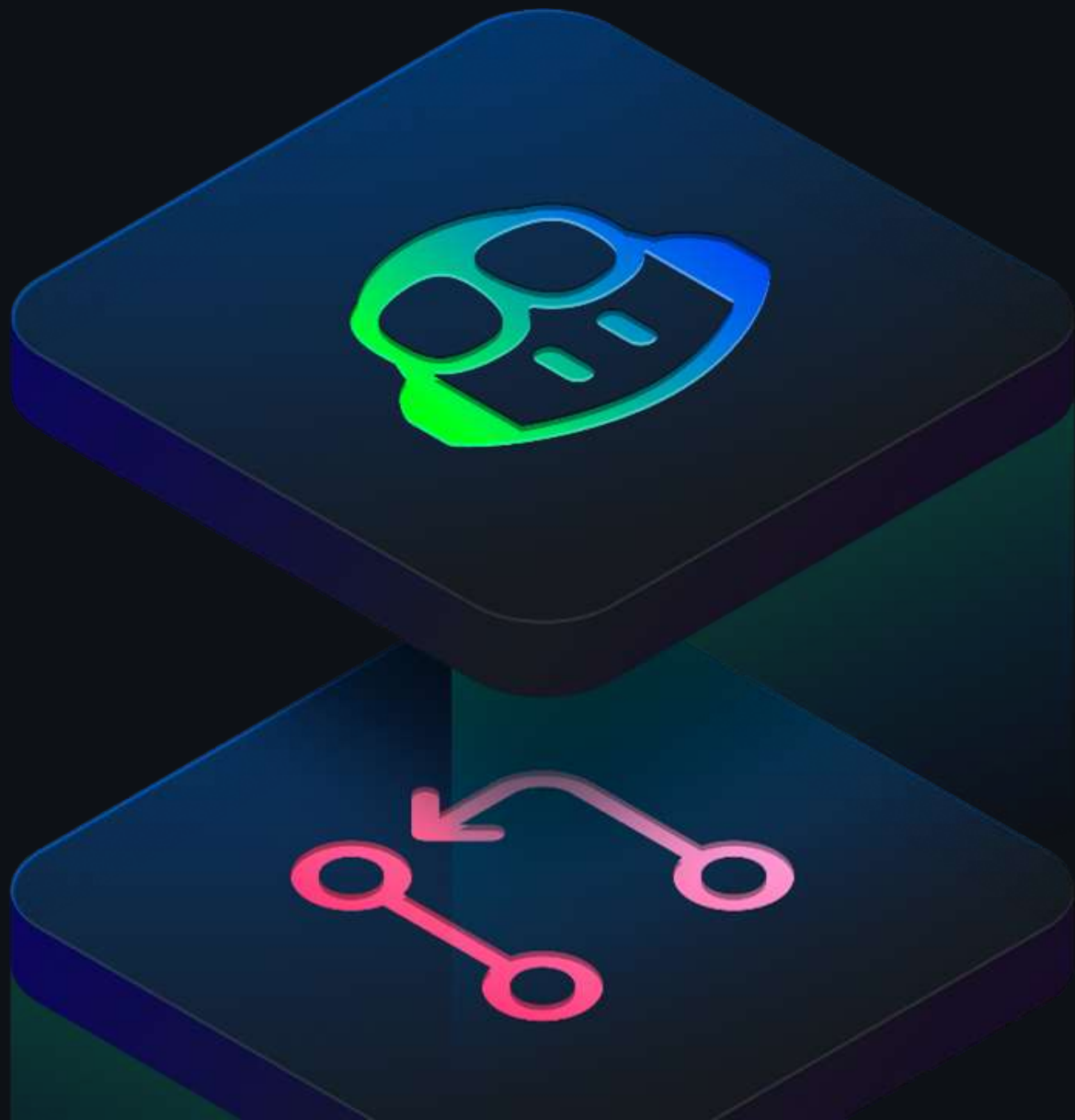




GitHub Copilot



GitHub Copilot: AI による開発者支援



開発者満足度を最大化

開発者生産性の向上

ソフトウェア開発を促進

各種開発環境サポート



Visual Studio
Code



Visual Studio
(Windows)



GitHub
Codespaces



JetBrains IDEs



Neovim

GitHub Copilot のデータフロー



ソフトウェア開発ライフサイクルと GitHub Copilot



実装フェーズ

● 実装

- ・ コメントをコードに変換
- ・ 反復コードの自動補完
- ・ 代替案の提示

実装フェーズ以外

● テスト & 統合

- ・ 単体テスト
- ・ コードエラー検知
- ・ デバッグ
- ・ コードレビュー
- ・ AI による Pull Request

● 保守

- ・ コードリファクタリング
(コードの翻訳)
- ・ コードレビュー
(コードの説明)
- ・ ドキュメント化

(参照) GitHub Copilot プラン

GitHub Copilot プラン





Step 1

GitHub Copilot Chat

Task 0: ワークショップ用のリポジトリ

```
> git clone https://github.com/shinyay/code-to-code-ws-20250205.git
```

Task 1: Copilot チャットで質問する

COBOL から Java への変換することを検討しています。

この変換プロジェクトにおいて、私のチームが移行する必要があるコードベース、モジュール、機能の範囲をどのように決定すればよいのでしょうか？また、どのような手順を考えたらよいのでしょうか？

(参照) Copilot Chat のプロンプトの概要

Copilot Chat のプロンプトの概要



Task 1: Copilot 回答例

COBOL から新しい言語への変換プロジェクトのコードベース、モジュール、および機能の範囲を決定するには、次の手順に従います:

1. COBOL コードベースのインベントリ:

1. すべての COBOL プログラム、スクリプト、および関連ファイルを特定します。
2. コードベースのサイズ (例: コード行数、ファイル数) を文書化します。

2. モジュールとコンポーネントの特定:

1. COBOL コードベースを論理的なモジュールとコンポーネントに分解します。
2. 各モジュールの目的と機能を文書化します。

3. 依存関係の分析:

1. COBOL モジュール間の依存関係を特定します。
2. 外部依存関係 (例: データベース、サードパーティライブラリ) を文書化します。

4. 機能の評価:

1. COBOL アプリケーションが提供するすべての機能をリストアップします。
2. ビジネスの重要性に基づいて機能を優先順位付けします。

5. データ構造の評価:

1. COBOL アプリケーションで 사용되는データ構造を文書化します。
2. データストレージメカニズム (例: ファイル、データベース) を特定します。

6. ビジネスロジックのレビュー:

1. COBOL コードに埋め込まれたビジネスルールとロジックを抽出して文書化します。

Task 1: Copilot 回答例

6. 統合ポイントの特定:

1. 他のシステムとの統合ポイントを特定します。
2. インターフェースと通信プロトコルを文書化します。

7. 作業量の見積もり:

1. 各モジュールと機能を変換するために必要な作業量を見積もります。
2. 複雑さ、サイズ、および依存関係を考慮します。

8. テストと検証の計画:

1. 変換された Java アプリケーションのテスト戦略を定義します。
2. 単体テスト、統合テスト、およびユーザー受け入れテストの計画を立てます。

9. 移行ロードマップの作成:

1. タイムラインとマイルストーンを含む詳細な移行計画を作成します。
2. リスク評価と軽減戦略を含めます。

これらの手順に従うことで、COBOL から Java への変換プロジェクトの範囲を包括的に決定し、構造化された効率的な移行プロセスを確保できます。

Task2: Copilot の回答に基づいた追加の質問

COBOL から新しい言語への変換作業と手順について確認できました。次に、各手順に関して詳細な内容について理解したいです。

COBOL コードベースのインベントリを作成する手順を詳細に説明してください。また、その手順を実施する中で GitHub Copilot がどのように支援できるか教えてください。

(参照) Copilot の長所と短所を理解する

Copilot の長所と短所を理解する



Task 2: Copilot 回答例

COBOL コードベースのインベントリを作成する手順は以下の通りです。

1. コードベースの収集

•すべての COBOL ファイルを収集:

- プロジェクトディレクトリ内のすべての COBOL ソースファイル (.cbl、.cob、.cpy など) を収集します。
- スクリプトやバッチファイルも含めます。

2. コードベースのサイズを測定

•コード行数のカウント:

- 各ファイルのコード行数をカウントします。
- コメント行と空行を除外するかどうかを決定します。

3. ファイルの分類

•ファイルの分類:

- プログラムファイル、コピー本 (COPYBOOK) 、JCL (Job Control Language) ファイルなどに分類します。
- 各ファイルの役割を特定します。

Task 2: Copilot 回答例

4. 依存関係の特定

・依存関係のマッピング:

- ・ 各プログラムが使用するコピー本や外部ファイルを特定します。
- ・ プログラム間の呼び出し関係をマッピングします。

5. ドキュメントの作成

・インベントリドキュメントの作成:

- ・ 収集した情報をドキュメントにまとめます。
- ・ 各ファイルのパス、サイズ、依存関係、役割などを記載します。

Task 3: コードからドキュメント / ドキュメントからコード への利点について Copilot に尋ねる

COBOL の変換プロジェクトについての質問です。

私はCOBOL を異なる言語に変換するツールがあることを知っています。

COBOL のコードベースを解釈して理解した後に GitHub Copilot を使用してコードを変換していく作業を行うことの利点は、COBOLコードの変換ツールを使用することと比較して何があるのでしょうか？

(参照) 思慮深いプロンプトを作成する

思慮深いプロンプトを作成する



Task 3: Copilot 回答例

COBOL コードベースの理解に基づいて GitHub Copilot を使用して COBOL コードを移行することには、自動変換ツールを使用することと比較していくつかの利点があります。以下に主要な利点を示します:

1. カスタマイズと柔軟性

- **カスタマイズされたソリューション:** GitHub Copilot を使用すると、汎用的な変換ルールに依存するのではなく、特定の要件やビジネスロジックに合わせたカスタムコードを作成できます。
- **柔軟性:** アプリケーションのニーズに合わせた設計上の決定や最適化を行うことができ、自動ツールでは不可能な場合があります。

2. コード品質と保守性

- **読みやすいコード:** GitHub Copilot で生成されたコードは、ターゲット言語の標準的なコーディングプラクティスや規約に従うため、読みやすく保守しやすいことが多いです。
- **リファクタリング:** 移行プロセス中にコードをリファクタリングして改善することができ、結果として得られるコードベースがクリーンで保守しやすくなります。

3. 理解とコントロール

- **深い理解:** GitHub Copilot を使用して手動でコードを移行することで、元の COBOL コードと新しいコードベースの両方を深く理解することができ、将来の保守や拡張に役立ちます。
- **移行のコントロール:** 移行プロセスを完全にコントロールでき、エッジケース、複雑なビジネスロジック、特定の要件を処理することができます。

Task 3: Copilot 回答例

4. エラーハンドリングとデバッグ

- カスタムエラーハンドリング:** 自動ツールが提供する汎用的なエラーハンドリングに依存するのではなく、アプリケーションに適したカスタムエラーハンドリングとロギングメカニズムを実装できます。
- デバッグ:** 手動でコードを移行することで、アプリケーションの各部分を徹底的にテストしてデバッグし、新しい環境で正しく動作することを確認できます。

5. 最新技術との統合

- 最新のプラクティス:** 移行プロセス中に最新の開発プラクティス、フレームワーク、およびライブラリを統合することができ、自動ツールでは不可能な場合があります。
- シームレスな統合:** データベース、API、サードパーティサービスなど、システムの他の部分とのシームレスな統合を確保できます。

6. 段階的な移行

- ステップバイステップの移行:** GitHub Copilot を使用すると、コードを段階的に移行し、各部分をテストして検証してから次に進むことができます。これにより、エラーのリスクが軽減され、移行プロセスが管理しやすくなります。
- 並行開発:** 既存の COBOL アプリケーションを引き続き開発および保守しながら、新しい言語への移行を段階的に進めることができます。

結論

自動変換ツールは大規模なコードベースを迅速に変換するのに役立ちますが、GitHub Copilot を使用して手動で COBOL コードを移行することで、カスタマイズ、コントロール、および結果として得られるコードの理解が向上します。このアプローチは、コードの品質、保守性、および最新の開発プラクティスや技術との統合を確保します。

Task 4: 自動変換ツールについて Copilot に尋ねる

COBOL の自動変換ツールを使用することの利点と欠点はどのようなことがあるでしょうか？詳細に教えてください。

(参照) Copilot を役立つ出力に導く

Copilot を役立つ出力に導く



Task 4: Copilot 回答例

COBOL の自動変換ツールを使用することの利点と欠点について、以下に詳細に説明します。

利点

1. 迅速な変換:

1. 自動変換ツールは一括でコードを変換するため、手動での変換に比べて非常に迅速に移行を進めることができます。
2. 大規模なコードベースでも短期間で変換が可能です。

2. 一貫性:

1. 変換ツールは一貫した変換ルールに基づいてコードを変換するため、変換後のコードに一貫性があります。
2. すべてのコードが同じスタイルとパターンで変換されるため、コードの可読性が向上します。

3. 専門知識の不要:

1. 変換ツールは COBOL とターゲット言語の専門知識がなくても使用できるため、開発者のスキルセットに依存しません。
2. 開発チーム全体での学習コストを削減できます。

4. エラーの削減:

1. 自動変換ツールは人為的なミスを減らすことができます。
2. 手動変換に比べて、変換中のエラーが少なくなります。

5. コスト削減:

1. 迅速な変換とエラーの削減により、プロジェクトの総コストを削減できます。
2. 手動での変換に比べて、開発時間とリソースを節約できます。

Task 4: Copilot 回答例

欠点

1. カスタマイズ性の欠如:

1. 自動変換ツールは一般的な変換ルールに基づいて動作するため、特定のプロジェクトやビジネスロジックに合わせたカスタマイズが難しい場合があります。
2. 特殊な要件や独自のビジネスロジックを持つコードの変換には不向きです。

2. コードの理解不足:

1. 自動変換ツールはコードの文脈やビジネスロジックを理解せずに変換を行うため、変換後のコードが最適化されていない場合があります。
2. ビジネスロジックの意図を正確に反映できないことがあります。

3. テストと検証の必要性:

1. 自動変換後のコードは、必ずしも完全に動作するわけではないため、徹底的なテストと検証が必要です。
2. 変換ツールが生成したコードにバグやパフォーマンスの問題が含まれている可能性があります。

4. 依存関係の問題:

1. 自動変換ツールは、外部依存関係やシステム間の統合ポイントを適切に処理できない場合があります。
2. これにより、変換後のシステムが正しく動作しないリスクがあります。

5. 学習曲線:

1. 自動変換ツールの使用方法や設定に関する学習が必要です。
2. ツールの使い方を習得するための時間とリソースが必要です。

Task 5: COBOL に関する専門用語

COBOLにはサブルーチンやコピー句などというようなCOBOL固有の専門用語が多数あると認識しています。COBOLを他の言語に変換することを考えるにあたり、知っておくべき専門用語についてリストアップしてそれぞれについて説明をしてください。

(参照) Copilot Chat にアクセスする方法

Copilot Chat にアクセスする方法



Task 5: Copilot 回答例

COBOL を他の言語に変換する際に知っておくべき専門用語についてリストアップし、それぞれについて説明します。

1. IDENTIFICATION DIVISION

プログラムの識別情報を記述するセクションです。プログラム名や作成者、日付などが含まれます。

2. ENVIRONMENT DIVISION

プログラムが実行される環境に関する情報を記述するセクションです。ファイルの割り当てや外部デバイスの指定などが含まれます。

3. DATA DIVISION

プログラムで使用するデータを定義するセクションです。以下のセクションに分かれます。

WORKING-STORAGE SECTION: プログラム内で使用される変数を定義します。

FILE SECTION: ファイルのレコードレイアウトを定義します。

LINKAGE SECTION: 外部プログラムから渡されるデータを定義します。

4. PROCEDURE DIVISION

プログラムの処理手順を記述するセクションです。実際のロジックやアルゴリズムが含まれます。

5. COPYBOOK

再利用可能なコードの断片を定義するファイルです。COPY ステートメントを使用してプログラムに挿入されます。

Task 5: Copilot 回答例

6. PERFORM

サブルーチンやループを実行するためのステートメントです。特定の処理を繰り返し実行する際に使用されます。

7. CALL

外部プログラムやサブルーチンを呼び出すためのステートメントです。引数を渡して処理を委譲します。

8. SECTION

PROCEDURE DIVISION 内で論理的に関連する一連のステートメントをグループ化するための単位です。

9. PARAGRAPH

SECTION 内のさらに小さな単位で、特定の処理を行う一連のステートメントをグループ化します。

10. PIC (PICTURE)

データ項目のフォーマットを定義するための句です。データの型やサイズを指定します。

例: PIC 9(5) は5桁の数字を表します。

11. OCCURS

配列を定義するための句です。特定のデータ項目が複数回繰り返されることを示します。

12. VALUE

データ項目の初期値を設定するための句です。

Task 5: Copilot 回答例

13. COMPUTE

算術演算を行い、その結果を変数に格納するためのステートメントです。

14. IF

条件分岐を行うためのステートメントです。条件に応じて異なる処理を実行します。

15. MOVE

データを別の変数にコピーするためのステートメントです。

16. STOP RUN

プログラムの実行を終了するためのステートメントです。

17. DISPLAY

画面にメッセージやデータを表示するためのステートメントです。

18. ACCEPT

ユーザーからの入力を受け取るためのステートメントです。

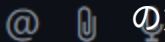
19. SORT

データを特定の順序で並べ替えるためのステートメントです。

20. SEARCH

配列内の特定の要素を検索するためのステートメントです。

これらの専門用語を理解することで、COBOL プログラムを他の言語に変換する際に役立ちます。各用語の意味と役割を把握することで、変換作業を効率的に進めることができます。



GPT 4o





Step 2

COBOLコードの分析

Task 1: コードの依存関係分析

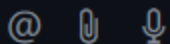
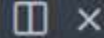
#file:main.cobc このコードを実行するために必要な依存関係を分析してください。

main.cobc

main.cobc single-file

recently opened

main.cobc multiple-file



GPT 4o

(参照) チャット変数

チャット変数



Task 1: Copilot 回答例

この COBOL コードを実行するために必要な依存関係を分析するには、次の手順に従います：

1. 内部依存関係:

- データ構造: コードは WORKING-STORAGE SECTION で定義されたさまざまなデータ構造に依存しています。
- サブルーチン: PROCEDURE DIVISION には MAIN-PROCEDURE 内で呼び出されるいくつかのサブルーチンが含まれています。

2. 外部依存関係:

- 呼び出されるプログラム: コードは CALL ステートメントを使用して外部の COBOL プログラム CalculateBonus を呼び出します。

詳細な分析:

内部依存関係:

1. データ構造:

- MAX-EMPLOYEES: 最大従業員数を定義します。
- EMPLOYEE-TABLE: 従業員の詳細を格納します。
- SORTED-EMPLOYEE-TABLE: ソートされた従業員の詳細を格納します。
- DEPARTMENT-TOTALS: 部門ごとの総給与を格納します。
- その他の変数として EMPLOYEE-INDEX, INNER-INDEX, TAX-RATE などがあります。

Task 1: Copilot 回答例

2. サブルーチン:

- INITIALIZE-EMPLOYEES: 従業員データを初期化します。
- SORT-EMPLOYEES: 従業員をIDでソートします。
- CALCULATE-NET-SALARIES: 従業員の純給与を計算します。
- CALCULATE-DEPARTMENT-TOTALS: 各部門の総給与を計算します。
- DISPLAY-EMPLOYEES: 従業員の給与情報を表示します。
- DISPLAY-DEPARTMENT-TOTALS: 部門ごとの給与総額を表示します。

外部依存関係:

1. 呼び出されるプログラム:

- CalculateBonus: このプログラムは各従業員のボーナスを計算するために呼び出されます。CALCULATE-NET-SALARIES サブルーチン内の CALL ステートメントがこの依存関係を示しています。

まとめ:

このコードを実行するには、次の依存関係が満たされていることを確認してください：

- すべてのデータ構造と変数が正しく定義および初期化されていること。
- すべての内部サブルーチン（INITIALIZE-EMPLOYEES, SORT-EMPLOYEES など）が実装されていること。
- 外部プログラム CalculateBonus が利用可能で正しく実装されていること。

@ 📎 これらの依存関係が満たされていることで、この プログラムは正常に実行できます。

GPT 4o ▾ ➤ ▾

Task 2: コードの説明

@workspace /explain #file:main.cobc
を詳細に説明してください。

(参照) チャット参加者/スラッシュコマンド

チャット参加者/スラッシュコマンド



Task 2: Copilot 回答例

COBOL コードを詳細に見ていきましょう。

識別部

このセクションはプログラムを識別します。

IDENTIFICATION DIVISION.

PROGRAM-ID. PayrollSystem.

データ部

このセクションはプログラムで使用するデータ構造と変数を定義します。

作業記憶セクション

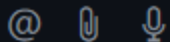
このセクションにはプログラム全体で使用する変数が含まれます。

DATA DIVISION.

WORKING-STORAGE SECTION.

01 MAX-EMPLOYEES PIC 9(3) VALUE 5.

MAX-EMPLOYEES は最大値999の数値変数で、初期値は5です。



Task 2: Copilot 回答例

従業員テーブル

このテーブルは従業員の詳細を格納します。

```
01 EMPLOYEE-TABLE.
```

```
05 EMPLOYEE-ENTRY OCCURS 5 TIMES.
```

```
10 EMPLOYEE-ID    PIC X(5).
```

```
10 EMPLOYEE-NAME  PIC X(20).
```

```
10 DEPARTMENT     PIC X(10).
```

```
10 GROSS-SALARY   PIC 9(7)V99.
```

```
10 BONUS          PIC 9(5)V99.
```

```
10 DEDUCTIONS     PIC 9(5)V99.
```

```
10 NET-SALARY     PIC 9(7)V99.
```

```
10 TAX-DEDUCTION  PIC 9(5)V99.
```

EMPLOYEE-ENTRY は5回繰り返され、それぞれのエントリには EMPLOYEE-ID, EMPLOYEE-NAME, DEPARTMENT, GROSS-SALARY, BONUS, DEDUCTIONS, NET-SALARY, TAX-DEDUCTION が含まれます。

Task 2: Copilot 回答例

ソートされた従業員テーブル

このテーブルはソートされた従業員の詳細を格納します。

```
01 SORTED-EMPLOYEE-TABLE.
```

```
05 SORTED-EMPLOYEE-ENTRY OCCURS 5 TIMES.
```

```
10 SORT-EMPLOYEE-ID PIC X(5).
```

```
10 SORT-EMPLOYEE-NAME PIC X(20).
```

```
10 SORT-DEPARTMENT PIC X(10).
```

```
10 SORT-GROSS-SALARY PIC 9(7)V99.
```

```
10 SORT-BONUS PIC 9(5)V99.
```

```
10 SORT-DEDUCTIONS PIC 9(5)V99.
```

```
10 SORT-NET-SALARY PIC 9(7)V99.
```

```
10 SORT-TAX-DEDUCTION PIC 9(5)V99.
```

EMPLOYEE-TABLE と似ていますが、ソートされたデータ用です。

Task 2: Copilot 回答例

部門合計

このテーブルは各部門の総給与を格納します。

```
01 DEPARTMENT-TOTALS.
```

```
05 DEPARTMENT-TOTAL OCCURS 5 TIMES.
```

```
10 DEPT-NAME      PIC X(10).
```

```
10 TOTAL-SALARY   PIC 9(7)V99.
```

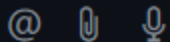
DEPARTMENT-TOTAL は5回繰り返され、それぞれのエントリには DEPT-NAME と TOTAL-SALARY が含まれます。

Task 2: Copilot 回答例

追加変数

これらの変数はさまざまな計算や一時的な格納に使用されます。

```
01 EMPLOYEE-INDEX    PIC 9(3).  
01 INNER-INDEX       PIC 9(3).  
01 TAX-RATE          PIC 9V99 VALUE 0.20.  
01 BONUS-RATE        PIC 9V99 VALUE 0.10.  
01 DEDUCTION-RATE    PIC 9V99 VALUE 0.05.  
01 DEPARTMENT-INDEX  PIC 9(3).  
  
01 TEMP-ID           PIC X(5).  
01 TEMP-NAME         PIC X(20).  
01 TEMP-DEPARTMENT   PIC X(10).  
01 TEMP-SALARY       PIC 9(7)V99.  
  
01 LNK-GROSS-SALARY  PIC 9(7)V99.  
01 LNK-BONUS         PIC 9(5)V99.
```



Task 2: Copilot 回答例

手続き部

このセクションには実行可能なコードが含まれます。

メイン手続き

プログラムのメインエントリポイントです。

PROCEDURE DIVISION.

MAIN-PROCEDURE.

PERFORM INITIALIZE-EMPLOYEES.

PERFORM SORT-EMPLOYEES.

PERFORM CALCULATE-NET-SALARIES.

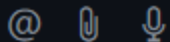
PERFORM CALCULATE-DEPARTMENT-TOTALS.

PERFORM DISPLAY-EMPLOYEES.

PERFORM DISPLAY-DEPARTMENT-TOTALS.

STOP RUN.

データの初期化、ソート、計算、表示を行うさまざまなサブルーチンを呼び出します。



Task 2: Copilot 回答例

従業員の初期化

従業員データを初期化します。

```
INITIALIZE-EMPLOYEES.
```

```
MOVE "E001" TO EMPLOYEE-ID(1).
```

```
MOVE "Alice Johnson" TO EMPLOYEE-NAME(1).
```

```
MOVE "HR" TO DEPARTMENT(1).
```

```
MOVE 70000.00 TO GROSS-SALARY(1).
```

```
...
```

```
MOVE "E005" TO EMPLOYEE-ID(5).
```

```
MOVE "Eve Davis" TO EMPLOYEE-NAME(5).
```

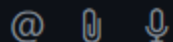
```
MOVE "HR" TO DEPARTMENT(5).
```

```
MOVE 75000.00 TO GROSS-SALARY(5).
```

ハードコードされた値を EMPLOYEE-TABLE に移動します。

従業員のソート

従業員をIDでソートします。



GPT 4o ▾ ➤ ▾

```
SORT-EMPLOYEES.
```

Task 2: Copilot 回答例

純給与の計算

各従業員の純給与を計算します。

```
CALCULATE-NET-SALARIES.
```

```
PERFORM VARYING EMPLOYEE-INDEX FROM 1 BY 1 UNTIL EMPLOYEE-INDEX > MAX-EMPLOYEES
```

```
    MOVE SORT-GROSS-SALARY(EMPLOYEE-INDEX) TO LNK-GROSS-SALARY
```

```
    CALL 'CalculateBonus' USING LNK-GROSS-SALARY LNK-BONUS
```

```
    MOVE LNK-BONUS TO SORT-BONUS(EMPLOYEE-INDEX)
```

```
    ...
```

```
END-PERFORM.
```

CalculateBonus サブルーチンを使用してボーナスを計算します。

控除、税金、純給与を計算します。

Task 2: Copilot 回答例

部門合計の計算

各部門の総給与を計算します。

```
CALCULATE-DEPARTMENT-TOTALS.
```

```
PERFORM VARYING DEPARTMENT-INDEX FROM 1 BY 1 UNTIL DEPARTMENT-INDEX > 5
```

```
    MOVE SPACES TO DEPT-NAME(DEPARTMENT-INDEX)
```

```
    MOVE 0 TO TOTAL-SALARY(DEPARTMENT-INDEX)
```

```
END-PERFORM.
```

```
...
```

```
PERFORM VARYING EMPLOYEE-INDEX FROM 1 BY 1 UNTIL EMPLOYEE-INDEX > MAX-EMPLOYEES
```

```
...
```

```
END-PERFORM.
```

部門合計を初期化します。

部門ごとに純給与を集計します。

Task 2: Copilot 回答例

従業員の表示

従業員の詳細を表示します。

```
DISPLAY-EMPLOYEES.
```

```
  DISPLAY "Employee Payroll Information".
```

```
  DISPLAY "-----".
```

```
  PERFORM VARYING EMPLOYEE-INDEX FROM 1 BY 1 UNTIL EMPLOYEE-INDEX > MAX-EMPLOYEES
```

```
    DISPLAY "Employee ID: " SORT-EMPLOYEE-ID(EMPLOYEE-INDEX)
```

```
  ...
```

```
END-PERFORM.
```

ソートされた従業員の詳細を表示します。

Task 2: Copilot 回答例

部門合計の表示

各部門の総給与を表示します。

```
DISPLAY-DEPARTMENT-TOTALS.  
  DISPLAY "Department Salary Totals".  
  DISPLAY "-----".  
  PERFORM VARYING DEPARTMENT-INDEX FROM 1 BY 1 UNTIL DEPARTMENT-INDEX > 5  
    IF DEPT-NAME(DEPARTMENT-INDEX) NOT = SPACES  
      DISPLAY "Department: " DEPT-NAME(DEPARTMENT-INDEX)  
    ...  
  END-IF  
END-PERFORM.
```

部門名が空でない場合に部門合計を表示します。

この COBOL プログラムは、従業員データを初期化し、ソートし、純給与を計算し、部門合計を集計し、結果を表示する給与システムです。

Task 3: コードの説明

```
#file:main.cobc  
#file:calculate_bonus.cobc このコード  
で定義されているサブルーチンは何で  
すか？リストアップし、それぞれにつ  
いて説明をしてください。
```

(参照) GitHub Copilot のプロンプト エンジニアリング

GitHub Copilot の
プロンプト エンジニアリング



Task 3: Copilot 回答例

この COBOL プログラムにはいくつかのサブルーチンが定義されています。以下にリストアップし、それぞれについて説明します。

サブルーチンのリスト

1. MAIN-PROCEDURE
2. INITIALIZE-EMPLOYEES
3. SORT-EMPLOYEES
4. CALCULATE-NET-SALARIES
5. CALCULATE-DEPARTMENT-TOTALS
6. DISPLAY-EMPLOYEES
7. DISPLAY-DEPARTMENT-TOTALS
8. CalculateBonus (外部プログラム)

各サブルーチンの説明

1. MAIN-PROCEDURE
 - プログラムのエントリーポイントであり、他のサブルーチンを順番に呼び出して全体の処理を制御します。
 - 実行される手順は、従業員データの初期化、ソート、給与計算、部門ごとの合計計算、従業員情報の表示、部門ごとの合計表示です。立したサブルーチンとして機能します。

Task 3: Copilot 回答例

2. INITIALIZE-EMPLOYEES

- ・ 従業員データを初期化するサブルーチンです。
- ・ 各従業員のID、名前、部門、基本給を設定します。

3. SORT-EMPLOYEES

- ・ 従業員データをソートするサブルーチンです。
- ・ 従業員IDに基づいて従業員データをソートし、ソートされたデータを SORTED-EMPLOYEE-TABLE に格納します。

4. CALCULATE-NET-SALARIES

- ・ 各従業員のネット給与を計算するサブルーチンです。
- ・ 基本給に対してボーナス、控除、税金を計算し、ネット給与を算出します。
- ・ 外部プログラム CalculateBonus を呼び出してボーナスを計算します。

5. CALCULATE-DEPARTMENT-TOTALS

- ・ 部門ごとの給与合計を計算するサブルーチンです。
- ・ 各部門の従業員のネット給与を合計し、部門ごとの合計給与を算出します。

6. DISPLAY-EMPLOYEES

- ・ 各従業員の給与情報を表示するサブルーチンです。
- ・ 従業員ID、名前、部門、基本給、ボーナス、控除、税金、ネット給与を表示します。能します。

Task 3: Copilot 回答例

7. DISPLAY-DEPARTMENT-TOTALS

- 部門ごとの給与合計を表示するサブルーチンです。
- 各部門の名前と合計給与を表示します。

8. CalculateBonus (外部プログラム)

- 外部プログラムとして定義されており、従業員のボーナスを計算するサブルーチンです。
- 基本給に対してボーナス率を掛けてボーナスを計算し、その結果を呼び出し元に返します。

まとめ

これらのサブルーチンは、COBOL プログラムの各機能を分割して実装するために使用されます。各サブルーチンは特定のタスクを実行し、プログラム全体の構造を明確にし、保守性を向上させます。外部プログラム CalculateBonus は、他のプログラムから呼び出される独立したサブルーチンとして機能します。

Task 3 - 2: コードの説明

#file:main.cobc
#file:calculate_bonus.cobc 各サブルーチン間の関係を詳細に説明し、それらのシーケンスについて説明をしてください。

Task 3 - 2: Copilot 回答例

以下に、main.cobc と calculate_bonus.cobc の各サブルーチン間の関係とシーケンスについて詳細に説明します。

main.cobc のサブルーチン

1. MAIN-PROCEDURE

- プログラムのエントリーポイントであり、他のサブルーチンを順番に呼び出して全体の処理を制御します。
- 呼び出されるサブルーチン:
 - INITIALIZE-EMPLOYEES, SORT-EMPLOYEES, CALCULATE-NET-SALARIES, CALCULATE-DEPARTMENT-TOTALS, DISPLAY-EMPLOYEES, DISPLAY-DEPARTMENT-TOTALS

2. INITIALIZE-EMPLOYEES

- 従業員データを初期化するサブルーチンです。
- 各従業員のID、名前、部門、基本給を設定します。

3. SORT-EMPLOYEES

- 従業員データをソートするサブルーチンです。
- 従業員IDに基づいて従業員データをソートし、ソートされたデータを SORTED-EMPLOYEE-TABLE に格納します。

4. CALCULATE-NET-SALARIES

- 各従業員のネット給与を計算するサブルーチンです。
- 基本給に対してボーナス、控除、税金を計算し、ネット給与を算出します。
- 外部プログラム CalculateBonus を呼び出してボーナスを計算します。ます。

Task 3 - 2: Copilot 回答例

5. CALCULATE-DEPARTMENT-TOTALS

- 部門ごとの給与合計を計算するサブルーチンです。
- 各部門の従業員のネット給与を合計し、部門ごとの合計給与を算出します。

6. DISPLAY-EMPLOYEES

- 各従業員の給与情報を表示するサブルーチンです。
- 従業員ID、名前、部門、基本給、ボーナス、控除、税金、ネット給与を表示します。

7. DISPLAY-DEPARTMENT-TOTALS

- 部門ごとの給与合計を表示するサブルーチンです。
- 各部門の名前と合計給与を表示します。

calculate_bonus.cobc のサブルーチン

1. CalculateBonus

- 外部プログラムとして定義されており、従業員のボーナスを計算するサブルーチンです。
- 基本給に対してボーナス率を掛けてボーナスを計算し、その結果を呼び出し元に返します。

特定のタスクを実行し、プログラム全体の処理フローを形成します。

Task 3 - 2: Copilot 回答例

サブルーチン間の関係とシーケンス

1. MAIN-PROCEDURE が開始されると、以下の順序でサブルーチンが呼び出されます:

- INITIALIZE-EMPLOYEES: 従業員データを初期化します。
- SORT-EMPLOYEES: 従業員データをソートします。
- CALCULATE-NET-SALARIES: 各従業員のネット給与を計算します。この中で、外部プログラム CalculateBonus が呼び出されます。
- CalculateBonus: CALCULATE-NET-SALARIES 内で CALL 'CalculateBonus' USING LNK-GROSS-SALARY LNK-BONUS により呼び出され、ボーナスを計算します。
- CALCULATE-DEPARTMENT-TOTALS: 部門ごとの給与合計を計算します。
- DISPLAY-EMPLOYEES: 各従業員の給与情報を表示します。
- DISPLAY-DEPARTMENT-TOTALS: 部門ごとの給与合計を表示します。

Task 3 - 2: Copilot 回答例

シーケンスの詳細

1. INITIALIZE-EMPLOYEES

- 従業員データを初期化します。各従業員のID、名前、部門、基本給を設定します。

2. SORT-EMPLOYEES

- 従業員データをソートします。まず、EMPLOYEE-TABLE のデータを SORTED-EMPLOYEE-TABLE にコピーし、その後、バブルソートアルゴリズムを使用して従業員IDに基づいてソートします。

3. CALCULATE-NET-SALARIES

- 各従業員のネット給与を計算します。まず、SORT-GROSS-SALARY を LNK-GROSS-SALARY に移動し、外部プログラム CalculateBonus を呼び出してボーナスを計算します。その後、控除と税金を計算し、ネット給与を算出します。

4. CALCULATE-DEPARTMENT-TOTALS

- 部門ごとの給与合計を計算します。各部門の従業員のネット給与を合計し、部門ごとの合計給与を算出します。

5. DISPLAY-EMPLOYEES

- 各従業員の給与情報を表示します。従業員ID、名前、部門、基本給、ボーナス、控除、税金、ネット給与を表示します。

6. DISPLAY-DEPARTMENT-TOTALS

- 部門ごとの給与合計を表示します。各部門の名前と合計給与を表示します。



Step 3

COBOLコードの変換調査

Task 1: コードの説明

COBOLプログラムをJavaに移行したいと考えています。

私は COBOL と Java はプログラム言語として異なる概念を持っていると認識しています。そのため、ソースコードを変換して移行する際にどのようなステップを踏むべきかを考える必要があると考えています。

例えば、Javaでクラスをどのように COBOL 言語から考えるべきか、COBOLのサブルーチンやコピー句を Java に移行する方法など、さまざまな考慮事項があると考えています。

そこで、COBOL から Java への移行手順について教えてください。

Task 1: Copilot 回答例

COBOL から Java への移行手順について、以下のステップを踏むことをお勧めします。これらのステップは、COBOL の概念を Java に適切にマッピングし、移行プロセスをスムーズに進めるためのものです。

1. プロジェクトの計画と準備

- コードベースのインベントリ:
 - すべての COBOL プログラム、コピー句、外部依存関係を特定し、ドキュメント化します。
- 移行の範囲と目標の設定:
- 移行するコードの範囲を決定し、移行の目標（例：パフォーマンスの向上、保守性の向上）を設定します。

2. COBOL の概念を Java にマッピング

- プログラム構造:
 - COBOL の各プログラムを Java のクラスにマッピングします。COBOL の IDENTIFICATION DIVISION は Java クラスの宣言に相当します。
- データ定義:
 - COBOL の DATA DIVISION に定義されたデータ項目を Java のフィールドに変換します。COBOL のデータ型を Java のデータ型にマッピングします。
- サブルーチン:
 - COBOL のサブルーチン（PERFORM や CALL ステートメント）を Java のメソッドに変換します。サブルーチンの引数はメソッドのパラメータとして扱います。
- COPYBOOK:
 - COBOL のコピー句（COPYBOOK）は、Java のクラスやインターフェースとして再利用可能なコードに変換します。

Task 1: Copilot 回答例

3. 移行の実装

- クラスの作成:
 - 各 COBOL プログラムに対応する Java クラスを作成します。クラス名はプログラム名に基づいて命名します。
- フィールドの定義:
 - COBOL のデータ項目を Java のフィールドとして定義します。データ型の変換を行います（例：PIC 9(3) は int に変換）
- メソッドの実装:
 - COBOL のサブルーチンを Java のメソッドとして実装します。メソッド名はサブルーチン名に基づいて命名します。
- 外部プログラムの呼び出し:
 - COBOL の外部プログラム呼び出し（CALL ステートメント）を Java のメソッド呼び出しに変換します。

4. テストと検証

- 単体テストの作成:
 - 各 Java クラスとメソッドに対して単体テストを作成し、正しく動作することを確認します。
- 統合テストの実施:
 - 移行された Java コード全体を統合テストし、システム全体が正しく動作することを確認します。

Task 1: Copilot 回答例

5. 最適化とリファクタリング

- コードの最適化:
 - 移行された Java コードを最適化し、パフォーマンスを向上させます。
- リファクタリング:
 - コードの可読性と保守性を向上させるために、リファクタリングを行います。

まとめ

COBOL から Java への移行は、プログラム構造、データ定義、サブルーチン、コピー句などの COBOL の概念を Java に適切にマッピングすることが重要です。移行プロセスを計画し、各ステップを慎重に実行することで、スムーズな移行を実現できます。

Task 2: COBOL to Java 移行手順

COBOL から Java への移行手順について確認ができました。
それでは、それぞれの移行手順の中で GitHub Copilot がどのように役立つことができるか教えてください。

Task 2: Copilot 回答例

GitHub Copilot は、COBOL から Java への移行手順の各ステップでさまざまな方法で役立つことができます。以下に、各移行手順において GitHub Copilot がどのように支援できるかを説明します。

1. プロジェクトの計画と準備

- コードベースのインベントリ:
 - GitHub Copilot は、スクリプトを生成して COBOL ファイルのリストを取得し、コード行数やファイルサイズを計算するのに役立ちます。
- 移行の範囲と目標の設定:
 - GitHub Copilot は、移行計画のドキュメントテンプレートを生成するのに役立ちます。

```
# Migration Plan
```

```
## Scope
```

- List of COBOL programs to be migrated
- Dependencies and external systems

```
## Goals
```

- Improve performance
- Enhance maintainability
- Reduce technical debt

```
## Timeline
```

- Phase 1: Inventory and Analysis
- Phase 2: Code Conversion
- Phase 3: Testing and Validation
- Phase 4: Optimization and Refactoring

Task 2: Copilot 回答例

2. COBOL の概念を Java にマッピング

- プログラム構造:

- GitHub Copilot は、COBOL プログラムを Java クラスに変換するためのテンプレートを生成するのに役立ちます。

```
public class PayrollSystem {  
    // Fields and methods will be added here  
}
```

- データ定義:

- GitHub Copilot は、COBOL のデータ項目を Java のフィールドに変換するコードを生成するのに役立ちます。

```
public class Employee {  
    private String id;  
    private String name;  
    private String department;  
    private double grossSalary;  
  
    // Constructor, getters, and setters  
}
```

Task 2: Copilot 回答例

- サブルーチン:
 - GitHub Copilot は、COBOL のサブルーチンを Java のメソッドに変換するコードを生成するのに役立ちます。

```
public void initializeEmployees() {  
    employees[0] = new Employee("E001", "Alice Johnson", "HR", 70000.00);  
    // Initialize other employees  
}
```

- COPYBOOK:
 - GitHub Copilot は、COBOL のコピー句を Java のクラスやインターフェースに変換するコードを生成するのに役立ちます。

```
public class EmployeeData {  
    // Fields and methods for reusable data structures  
}
```

3. 移行の実装

- クラスの作成:
 - GitHub Copilot は、COBOL プログラムに対応する Java クラスを作成するコードを生成するのに役立ちます。

```
public class PayrollSystem {  
    private Employee[] employees = new Employee[5];  
    // Methods will be added here  
}
```

Task 2: Copilot 回答例

- フィールドの定義:
 - GitHub Copilot は、COBOL のデータ項目を Java のフィールドに変換するコードを生成するのに役立ちます。

```
private int employeeIndex;  
private double taxRate = 0.20;  
private double bonusRate = 0.10;  
private double deductionRate = 0.05;
```

- メソッドの実装:
 - GitHub Copilot は、COBOL のサブルーチンを Java のメソッドに変換するコードを生成するのに役立ちます。

```
public void calculateNetSalaries() {  
    for (Employee employee : employees) {  
        double bonus = calculateBonus(employee.getGrossSalary());  
        double netSalary = employee.getGrossSalary() + bonus -  
            (employee.getGrossSalary() * taxRate) - (employee.getGrossSalary() *  
            deductionRate);  
        employee.setNetSalary(netSalary);  
    }  
}
```

Task 2: Copilot 回答例

- 外部プログラムの呼び出し:
 - GitHub Copilot は、COBOL の外部プログラム呼び出しを Java のメソッド呼び出しに変換するコードを生成するのに役立ちます。

```
public void callCalculateBonus(double grossSalary, double[] bonus) {  
    bonus[0] = calculateBonus(grossSalary);  
}
```

4. テストと検証

- 単体テストの作成:
 - GitHub Copilot は、各 Java クラスとメソッドに対する単体テストを生成するのに役立ちます。

```
@Test  
public void testInitializeEmployees() {  
    PayrollSystem payrollSystem = new PayrollSystem();  
    payrollSystem.initializeEmployees();  
    assertEquals("E001", payrollSystem.getEmployees()[0].getId());  
}
```


Task 2: Copilot 回答例

- 統合テストの実施:
 - GitHub Copilot は、統合テストのテンプレートを生成するのに役立ちます。

```
@Test
public void testPayrollSystemIntegration() {
    PayrollSystem payrollSystem = new PayrollSystem();
    payrollSystem.initializeEmployees();
    payrollSystem.calculateNetSalaries();
    payrollSystem.calculateDepartmentTotals();
    payrollSystem.displayEmployees();
    payrollSystem.displayDepartmentTotals();
}
```

5. 最適化とリファクタリング

- コードの最適化:
 - GitHub Copilot は、パフォーマンスを向上させるためのコード最適化の提案を行うのに役立ちます。

```
public void optimizeNetSalaryCalculation() {
    for (Employee employee : employees) {
        double grossSalary = employee.getGrossSalary();
        double bonus = calculateBonus(grossSalary);
        double netSalary = grossSalary + bonus - (grossSalary * taxRate) -
(grossSalary * deductionRate);
        employee.setNetSalary(netSalary); }}
```

Task 2: Copilot 回答例

- リファクタリング:
 - GitHub Copilot は、コードの可読性と保守性を向上させるためのリファクタリングの提案を行うのに役立ちます。

```
public class Employee {  
    private String id;  
    private String name;  
    private String department;  
    private double grossSalary;  
    private double netSalary;  
  
    // Constructor, getters, and setters  
}
```

まとめ

GitHub Copilot は、COBOL から Java への移行プロセスの各ステップで支援することができます。コードの生成、最適化、リファクタリング、テストの作成など、さまざまなタスクを効率的に行うための提案を提供し、移行プロジェクトをスムーズに進めることができます。

Task 3: Java 選択の理由

COBOL から Java への移行手順について確認ができました。

では、Java 以外の言語に対して COBOL を移行することを考えてみたいと思います。

その場合、まずどのようなプログラム言語が考えられるでしょうか？ またその言語が COBOL からの移行先として向いていると考えられる理由について説明してください。

Task 3: Copilot 回答例

COBOL からの変換先として Java が選ばれる理由とその利点について説明します。理由

1. オブジェクト指向プログラミング (OOP) のサポート:

- Java は完全なオブジェクト指向プログラミング言語であり、COBOL の構造化プログラミングからオブジェクト指向への移行を容易にします。
- OOP の概念（クラス、オブジェクト、継承、ポリモーフィズムなど）を活用することで、コードの再利用性と保守性が向上します。

2. クロスプラットフォームの互換性:

- Java は「Write Once, Run Anywhere (WORA)」の理念に基づいて設計されており、Java 仮想マシン (JVM) 上で動作するため、異なるプラットフォーム間での移植性が高いです。
- これにより、COBOL プログラムを Java に移行した後、さまざまなオペレーティングシステムで実行できます。

3. 豊富なライブラリとフレームワーク:

- Java には豊富な標準ライブラリとサードパーティのライブラリがあり、データベースアクセス、ネットワーク通信、ファイル操作、GUI 開発など、さまざまな機能を簡単に実装できます。
- Spring や Hibernate などの強力なフレームワークを使用することで、エンタープライズアプリケーションの開発が効率化されます。

4. 大規模なコミュニティとサポート:

- Java は広範なコミュニティとサポートがあり、豊富なドキュメントやリソースが利用可能です。
- 問題解決やベストプラクティスの共有が容易であり、移行プロジェクトをスムーズに進めることができます。

Task 3: Copilot 回答例

5. パフォーマンスとスケーラビリティ:

1. Java は高いパフォーマンスとスケーラビリティを提供し、大規模なエンタープライズアプリケーションの要件を満たすことができます。
2. JVM の最適化とガベージコレクションにより、効率的なメモリ管理とパフォーマンス向上が図られます。

利点

1.保守性の向上:

1. オブジェクト指向設計により、コードのモジュール化と再利用性が向上し、保守性が高まります。
2. クラスやメソッドの分割により、コードの理解と変更が容易になります。

2.セキュリティ:

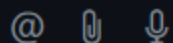
1. Java はセキュリティ機能が強化されており、セキュアなアプリケーション開発が可能です。
2. JVM のセキュリティマネージャーやサンドボックス機能により、アプリケーションの安全性が確保されます。

3.エンタープライズアプリケーションのサポート:

1. Java はエンタープライズアプリケーションの開発に適しており、JEE (Java Enterprise Edition) を使用することで、トランザクション管理、セッション管理、メッセージングなどの機能を簡単に実装できます。

4.豊富なツールとIDE:

1. Eclipse、IntelliJ IDEA、NetBeans などの強力な統合開発環境 (IDE) が利用可能であり、開発効率が向上します。



2. デバッグ、テスト、自動化ツールのサポートにより、開発プロセスが効率化されます。GPT 4o



Task 3: Copilot 回答例

まとめ

COBOL から Java への移行は、オブジェクト指向プログラミングのサポート、クロスプラットフォームの互換性、豊富なライブラリとフレームワーク、大規模なコミュニティとサポート、高いパフォーマンスとスケーラビリティなどの理由から選ばれます。これにより、保守性の向上、セキュリティの強化、エンタープライズアプリケーションのサポート、豊富なツールとIDEの利用が可能となり、移行プロジェクトを成功に導くことができます。

Task 4: COBOL to Java以外の言語 移行手順

COBOL から Java への移行手順について確認ができました。

では、Java 以外の言語に対してCOBOL を移行することを考えてみたいと思います。

その場合、まずどのようなプログラム言語が考えられるでしょうか？またその言語がCOBOLからの移行先として向いていると考えられる理由について説明してください。

Task 4: Copilot 回答例

COBOL からの移行先として考えられるプログラム言語はいくつかあります。それぞれの言語がCOBOLからの移行先として向いている理由について説明します。

1. Python

理由:

- **シンプルで読みやすい構文:** Python はシンプルで読みやすい構文を持ち、COBOL のようなビジネスロジックを記述するのに適しています。
- **豊富なライブラリ:** データ処理やファイル操作、ネットワーク通信など、さまざまなライブラリが豊富に揃っており、COBOL の機能を簡単に再現できます。
- **広範なコミュニティサポート:** Python は広範なコミュニティサポートがあり、移行時の問題解決が容易です。

2. C#

理由:

- **強力なオブジェクト指向機能:** C# は強力なオブジェクト指向機能を持ち、COBOL の構造化プログラミングをオブジェクト指向に移行するのに適しています。
- **.NET エコシステム:** C# は .NET エコシステムの一部であり、Windows 環境での開発や運用に強みがあります。
- **豊富なツールとサポート:** Visual Studio などの強力な開発ツールがあり、移行プロセスを支援します。

Task 4: Copilot 回答例

3. JavaScript (Node.js)

理由:

- 非同期処理のサポート**: Node.js は非同期処理をサポートしており、リアルタイムアプリケーションの開発に適しています。
- クロスプラットフォーム**: JavaScript はクロスプラットフォームで動作し、Webアプリケーションやサーバーサイドアプリケーションの開発に適しています。
- 豊富なパッケージ**: npm による豊富なパッケージがあり、COBOL の機能を再現するためのライブラリが多数存在します。

4. Ruby

理由:

- シンプルで直感的な構文**: Ruby はシンプルで直感的な構文を持ち、COBOL のビジネスロジックを簡単に移行できます。
- 強力なフレームワーク**: Ruby on Rails などの強力なフレームワークがあり、Webアプリケーションの開発に適しています。
- 活発なコミュニティ**: Ruby は活発なコミュニティがあり、移行時のサポートが充実しています。

Task 4: Copilot 回答例

5. Go

理由:

- 高いパフォーマンス:** Go はコンパイル言語であり、高いパフォーマンスを発揮します。COBOL のパフォーマンス要件を満たすことができます。
- シンプルな構文:** Go はシンプルな構文を持ち、学習コストが低いです。
- 並行処理のサポート:** Go は並行処理をサポートしており、マルチスレッドアプリケーションの開発に適しています。

まとめ

COBOL からの移行先としては、Python、C#、JavaScript (Node.js)、Ruby、Go などが考えられます。それぞれの言語は、COBOL のビジネスロジックやデータ処理の要件を満たすための特長を持っています。移行先の選定は、プロジェクトの要件やチームのスキルセットに応じて行うことが重要です。



Step 4

設計書作成

Task 1: 設計書テンプレートの作成

COBOLプログラムの設計書を作成したいです。

まず、以下のようなセクションを含む設計書のテンプレートを提供してください。また、下記のような項目に加えてCOBOLプログラムの設計書として必要な項目があれば、それについても提供してください:

- はじめに
- システム概要
- 機能要件
- 非機能要件
- システムアーキテクチャ
- データモデル
- ユーザーインターフェース設計
- エラーハンドリングとログ
- テスト戦略
- デプロイメント計画

この設計ドキュメントのテンプレートを提供してください。

Task 1: Copilot 回答例

COBOL Program Design Document

はじめに

- 目的: このドキュメントは、COBOLプログラムの設計を詳細に説明するためのものです。
- 対象読者: 開発者、プロジェクトマネージャー、テスター、運用担当者
- 範囲: プログラムの機能、非機能要件、システムアーキテクチャ、データモデル、ユーザーインターフェース設計、エラーハンドリング、テスト戦略、デプロイメント計画

システム概要

- システム名:
- システムの目的:
- システムの概要:
- 主要な機能:

機能要件

- 機能1:
 - 説明:
 - 入力:
 - 出力:
- 機能2:
 - :

Task 1: Copilot 回答例

非機能要件

- ・パフォーマンス要件:
- ・信頼性要件:
- ・可用性要件:
- ・セキュリティ要件:
- ・保守性要件:

システムアーキテクチャ

- ・アーキテクチャ概要:
- ・コンポーネント図:
- ・データフロー図:

データモデル

- ・データベース設計:
 - ・ テーブル定義:
 - ・ 主キーと外部キー:
- ・データ項目定義:
 - ・ 項目名:
 - ・ データ型:
 - ・ 説明:

Task 1: Copilot 回答例

ユーザーインターフェース設計

•画面設計:

- 画面1:
 - 説明:
 - 入力項目:
 - 出力項目:

•操作フロー:

エラーハンドリングとログ

•エラーハンドリングポリシー:

•エラーメッセージ:

•ログ出力:

- ログの種類:
- ログフォーマット:

Task 1: Copilot 回答例

テスト戦略

- テスト計画:
- テストケース:
 - テストケース1:
 - 説明:
 - 入力:
 - 期待される出力:

• テスト環境:

デプロイメント計画

• デプロイメント手順:

• 環境設定:

• リリース管理:

Task 1: Copilot 回答例

追加項目（COBOLプログラム特有の項目）

•COBOLプログラム構造:

- IDENTIFICATION DIVISION:
- ENVIRONMENT DIVISION:
- DATA DIVISION:
- PROCEDURE DIVISION:

•COPYBOOKの使用:

- 使用するCOPYBOOKの一覧:
- 各COPYBOOKの説明:

•外部プログラムとの連携:

- 呼び出す外部プログラム:
- 呼び出される外部プログラム:

•バッチ処理の設計:

- バッチジョブのフロー:
- スケジューリング:

まとめ

•今後の課題:

•参考文献:

•付録:

Task 2: 設計書の作成

設計書テンプレート #file:designdoc-template.md に基づいて、
#file:main.cobc の設計書を作成してください。

Task 2: Copilot 回答例

main.cobcの設計ドキュメント

1. はじめに

- 目的: このドキュメントは、COBOL給与システムの詳細な設計を提供し、その機能、アーキテクチャ、およびデプロイメント計画を概説します。
- 範囲: このドキュメントは、従業員データの初期化、ソート、計算、および表示をカバーします。
- 対象読者: このドキュメントは、給与システムの開発および保守に関与する開発者、テスター、およびシステム管理者を対象としています。

2. システム概要

- システム説明: 給与システムは、従業員データを初期化し、従業員をソートし、純給与を計算し、部門ごとの合計を計算し、結果を表示します。
- 目的: 主な目的は、給与処理を自動化し、正確な給与計算を確保し、明確な給与情報を提供することです。

3. 機能要件

- 従業員データの初期化: ハードコードされた値で従業員データを初期化します。
- 従業員のソート: バブルソートアルゴリズムを使用して従業員をIDでソートします。
- 純給与の計算: ボーナス、控除、および税控除を計算して純給与を計算します。
- 部門ごとの合計計算: 純給与を集計して各部門の給与合計を計算します。
- 情報の表示: コンソールに従業員の給与情報と部門ごとの給与合計を表示します。

Task 2: Copilot 回答例

4. 非機能要件

- パフォーマンス: システムは最大1000人の従業員の給与を5秒以内に処理する必要があります。
- スケーラビリティ: システムは、追加の従業員および部門に対応できるようにスケーラブルである必要があります。
- 保守性: コードはよく文書化され、可読性と保守性のためにコーディング標準に従う必要があります。
- 信頼性: システムは99.9%の稼働時間を持ち、エラーを適切に処理する必要があります。

5. システムアーキテクチャ

- 高レベルアーキテクチャ: システムは、初期化、ソート、計算、および表示のサブルーチンを呼び出すメインプログラムで構成されています。
- コンポーネント: 主なコンポーネントは、従業員テーブル、ソートされた従業員テーブル、部門ごとの合計、および各機能のサブルーチンです。

Task 2: Copilot 回答例

6. データモデル

データ構造

- EMPLOYEE-TABLE
 - EMPLOYEE-ID: PIC X(5)
 - EMPLOYEE-NAME: PIC X(20)
 - DEPARTMENT: PIC X(10)
 - GROSS-SALARY: PIC 9(7)V99
 - BONUS: PIC 9(5)V99
 - DEDUCTIONS: PIC 9(5)V99
 - NET-SALARY: PIC 9(7)V99
 - TAX-DEDUCTION: PIC 9(5)V99
- SORTED-EMPLOYEE-TABLE
 - SORT-EMPLOYEE-ID: PIC X(5)
 - SORT-EMPLOYEE-NAME: PIC X(20)
 - SORT-DEPARTMENT: PIC X(10)
 - SORT-GROSS-SALARY: PIC 9(7)V99
 - SORT-BONUS: PIC 9(5)V99
 - SORT-DEDUCTIONS: PIC 9(5)V99
 - SORT-NET-SALARY: PIC 9(7)V99
 - SORT-TAX-DEDUCTION: PIC 9(5)V99

Task 2: Copilot 回答例

- DEPARTMENT-TOTALS
 - DEPT-NAME: PIC X(10)
 - TOTAL-SALARY: PIC 9(7)V99

インデックスとレート

- EMPLOYEE-INDEX: PIC 9(3)
- INNER-INDEX: PIC 9(3)
- TAX-RATE: PIC 9V99 VALUE 0.20
- BONUS-RATE: PIC 9V99 VALUE 0.10
- DEDUCTION-RATE: PIC 9V99 VALUE 0.05
- DEPARTMENT-INDEX: PIC 9(3)

一時変数

- TEMP-ID: PIC X(5)
- TEMP-NAME: PIC X(20)
- TEMP-DEPARTMENT: PIC X(10)
- TEMP-SALARY: PIC 9(7)V99

Task 2: Copilot 回答例

リンクセクション変数

- LNK-GROSS-SALARY: PIC 9(7)V99
- LNK-BONUS: PIC 9(5)V99

データフロー

- 初期化: 従業員データはハードコードされた値で初期化されます。
- ソート: 従業員データはIDでソートされ、ソートされた従業員テーブルに格納されます。
- 純給与の計算: 純給与は、基本給与、ボーナス、控除、および税控除を使用して計算されます。
- 部門ごとの合計計算: 各部門の給与合計は、純給与を集計して計算されます。
- 表示: コンソールに従業員の給与情報と部門ごとの給与合計が表示されます。

7. ユーザーインターフェース設計

- コンソール出力: システムは、コンソールに従業員の給与情報と部門ごとの給与合計を表示します。
- ユーザーインタラクション: ユーザーはプログラムを実行し、コンソール出力を表示することでシステムと対話します。

8. エラーハンドリングとログ

- エラーハンドリング: システムは入力データを検証し、処理中のエラーを処理します。
- ログ: システムは、トラブルシューティングのために主要なイベントとエラーをログファイルに記録します。

Task 2: Copilot 回答例

9. テスト戦略

- 単体テスト: 各サブルーチンは、正しい機能を確保するために個別にテストされます。
- 統合テスト: システム全体がテストされ、すべてのコンポーネントが連携して動作することを確認します。
- ユーザー受け入れテスト: エンドユーザーがシステムを検証し、要件を満たしていることを確認します。

10. デプロイメント計画

- デプロイメント手順:
 1. COBOLプログラムをコンパイルします。
 2. 実行可能ファイルをターゲット環境にデプロイします。
 3. システムが正しく機能していることを確認するために初期テストを実行します。
- 環境設定: ターゲット環境には、COBOLコンパイラとランタイムがインストールされている必要があります。
- ロールバック計画: デプロイメントの問題が発生した場合、システムの以前のバージョンに戻します。

Task 3: コードコメント追加

設計書 #file:design-doc_main.md をもとに、#file:main.cobc
の可読性を向上させるようにコメントを挿入してください。

(オプション)Task 4: シーケンスダイアグラムの作成

@mermaid /sequence #file:main.cobc のシーケンスを生成してください。

(参照) vscode-mermAlD

vscode-mermAlD





Step 5

Java クラス設計

Task 1: 変換基準の確認と定義

COBOLコードをJavaに再設計する際に考慮するクラス設計の基準は何ですか？

Task 1: Copilot 回答例

COBOLコードをJavaに再設計する際、結果のコードが保守可能で効率的であり、Javaのオブジェクト指向機能を活用するために、いくつかの基準を考慮する必要があります。以下は主な基準です:

1. カプセル化

- データ隠蔽: プライベートフィールドを使用し、データへのアクセスを制御するためにパブリックなゲッターとセッターを提供します。
- モジュール性: 関連するデータと動作をクラスにグループ化して、モジュール性と再利用性を促進します。

2. 単一責任の原則

- 凝集性: 各クラスが単一の責任を持ち、関連する機能をカプセル化するようにします。
- 関心の分離: 異なる関心事を異なるクラスに分離します (例: データ表現、ビジネスロジック、データアクセス)。

3. 継承とポリモーフィズム

- 階層: 適切な場合に継承を使用してクラス階層を作成し、コードの再利用性と拡張性を促進します。
- インターフェースと抽象クラス: 共通の動作を表すインターフェースや抽象クラスを定義し、ポリモーフィックな動作を可能にします。

4. 継承よりもコンポジション

- コンポジション: 継承よりもコンポジションを優先してコードの再利用性と柔軟性を実現します。クラス間の関係を表すためにメンバーオブジェクトを使用します。

Task 1: Copilot 回答例

5. データ構造

- コレクション: 適切なJavaコレクション（例: List, Map, Set）を使用してオブジェクトのグループを管理します。
- データ型: COBOLのデータ型を適切なJavaデータ型（例: String, int, double）にマッピングします。

6. エラーハンドリング

- 例外: Javaの例外処理メカニズムを使用してエラーや例外的な状況を管理します。
- バリデーション: 入力のバリデーションとエラーチェックを実装してデータの整合性を確保します。

7. スケーラビリティとパフォーマンス

- 効率性: メモリと処理時間の観点から効率的なクラスとメソッドを設計します。
- スケーラビリティ: 増加する負荷と複雑さに対応できる設計を行い、パフォーマンスの大幅な低下を防ぎます。

8. 保守性

- 可読性: 意味のあるクラス名とメソッド名を使用して、明確で読みやすいコードを書きます。
- ドキュメント: クラスとメソッドの目的と使用方法を説明するドキュメントとコメントを提供します。

9. テスト可能性

- ユニットテスト: クラスとメソッドがユニットテストで簡単にテストできるように設計します。
- モック: テストでモックを使用できるようにインターフェースと依存性注入を使用します。

Task 2: 設計内容に基づいたクラス設計定義

設計ドキュメント #file:design-doc_main.md と変換基準
#file:java-conversion-criteria.md に基づいて、
#file:main.cobc をJavaコードに変換する際に必要なクラスは
何ですか？

クラスとその決定理由を教えてください。

Task 2: Copilot 回答例

設計ドキュメントと変換基準に基づいて、main.cobcのCOBOLコードをJavaに変換する際に必要なクラスを以下に示します:

1. Employeeクラス

- 理由: ID、名前、部門、総給与、ボーナス、控除、純給与、税控除などの属性を持つ従業員を表します。このクラスは従業員に関連するデータをカプセル化し、データのアクセスと変更のためのメソッドを提供します。

2. DepartmentTotalクラス

- 理由: 部門名と総給与などの属性を持つ部門の総給与を表します。このクラスは部門の給与合計に関連するデータをカプセル化し、データのアクセスと変更のためのメソッドを提供します。

3. PayrollSystemクラス

- 理由: 従業員の初期化、従業員のソート、純給与の計算、部門の合計の計算、情報の表示などのメソッドを含みます。このクラスは全体の給与システムを管理し、さまざまなタスクの実行を調整します。

4. BonusCalculatorクラス

- 理由: CalculateBonus COBOLプログラムの機能を表します。このクラスには総給与に基づいてボーナスを計算するメソッドが含まれています。ボーナス計算のロジックをカプセル化し、PayrollSystemクラスから呼び出すことができます。

Task 2: Copilot 回答例

まとめ

- Employeeクラス: 従業員データをカプセル化し、データのアクセスと変更のためのメソッドを提供します。
- DepartmentTotalクラス: 部門の総給与データをカプセル化します。
- PayrollSystemクラス: 従業員の初期化、ソート、給与計算、情報表示など、給与システム全体を管理します。
- BonusCalculatorクラス: 総給与に基づいてボーナスを計算するロジックをカプセル化します。
- これらのクラスは、元のCOBOLプログラムの機能を保持しながら、Javaのオブジェクト指向機能を活用することを保証します。

Task 3: Java クラス詳細設計

設計ドキュメント #file:design-doc_main.md と変換基準
#file:java-conversion-criteria.md に基づいて、Employee クラ
スの詳細なクラス設計を提供してください。

Task 3: Copilot 回答例

Employeeクラスの詳細なクラス設計

設計ドキュメントと変換基準に基づいて、JavaのEmployeeクラスの詳細なクラス設計を以下に示します:

1. クラス定義

- クラス名: Employee
- 目的: ID、名前、部門、総給与、ボーナス、控除、純給与、税控除などの属性を持つ従業員を表します。

2. 属性

- id: String
 - 従業員のIDを表します。
- name: String
 - 従業員の名前を表します。
- department: String
 - 従業員が所属する部門を表します。
- grossSalary: double
 - 従業員の総給与を表します。
- bonus: double
 - 従業員のボーナスを表します。
- deductions: double
 - 従業員の控除を表します。
- netSalary: double
 - 従業員の純給与を表します。

Task 3: Copilot 回答例

3. コンストラクタ

- `Employee(String id, String name, String department, double grossSalary)`
- 指定されたID、名前、部門、および総給与で従業員を初期化します。他の属性はデフォルト値に初期化されます。

4. メソッド

- ゲッターとセッター
 - `String getId()`
 - `void setId(String id)`
 - `String getName()`
 - `void setName(String name)`
 - `String getDepartment()`
 - `void setDepartment(String department)`
 - `double getGrossSalary()`
 - `void setGrossSalary(double grossSalary)`
 - `double getBonus()`
 - `void setBonus(double bonus)`
 - `double getDeductions()`
 - `void setDeductions(double deductions)`
 - `double getNetSalary()`
 - `void setNetSalary(double netSalary)`
 - `double getTaxDeduction()`
 - `void setTaxDeduction(double taxDeduction)`

Task 3: Copilot 回答例

5. カプセル化

- すべての属性はカプセル化を確保するためにプライベートです。
- 属性にアクセスし変更するためのパブリックなゲッターとセッターが提供されます。

6. 保守性

- クラスは明確で読みやすく、意味のある属性名を持つように設計されています。
- クラスとそのメソッドの目的と使用方法を説明するドキュメントとコメントが提供されます。

7. テスト可能性

- クラスはユニットテストで簡単にテストできるように設計されています。
- メソッドは個別にテストして正しい機能を確認できます。

[cobol-sample-app/java/Employee.java at main · shinyay/cobol-sample-app](#)

まとめ

- カプセル化: すべての属性はプライベートであり、パブリックなゲッターとセッターが提供されます。
- 単一責任の原則: クラスは従業員データの表現にのみ焦点を当てています。
- 保守性: クラスは明確で読みやすく、よく文書化されています。
- テスト可能性: クラスはユニットテストで簡単にテストできるように設計されています。

この詳細なクラス設計により、Employeeクラスが堅牢で保守可能であり、Javaのオブジェクト指向機能を活用できることが保証されます。

(オプション) Task 4 - 6: Java クラス詳細設計

Task 4

設計ドキュメント #file:design-doc_main.md と変換基準 #file:java-conversion-criteria.md に基づいて、DepartmentTotalクラスの詳細なクラス設計を提供してください。

Task 5

設計ドキュメント #file:design-doc_main.md と変換基準 #file:java-conversion-criteria.md に基づいて、PayrollSystemクラスの詳細なクラス設計を提供してください。

Task 6

設計ドキュメント #file:design-doc_main.md と変換基準 #file:java-conversion-criteria.md に基づいて、BonusCalculatorクラスの詳細なクラス設計を提供してください。



Step 6

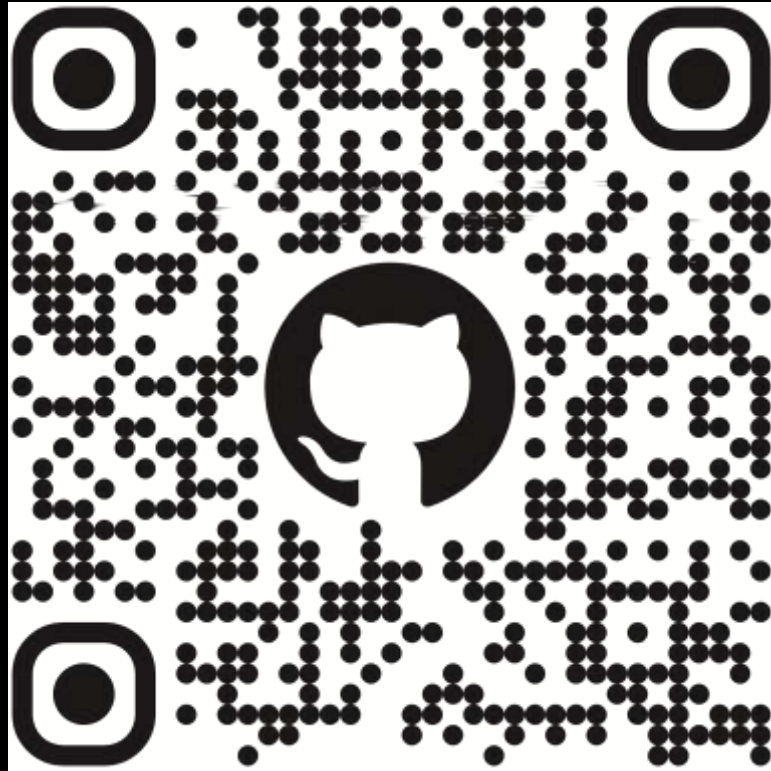
Java コード生成

Task 1: 設計書からコード生成

設計ドキュメント #file:employee-class.md に基づいて、java ディレクトリにEmployeeクラスを提案してください。

(参照) Copilot Edits

Copilot Edits



(オプション) Task 2 - 4: 設計書からコード生成

Task 2

設計ドキュメント #file:departmenttotal-class.md に基づいて、javaディレクトリにDepartmentTotalクラスを提案してください。

Task 3

設計ドキュメント #file:payrollsystem-class.md に基づいて、javaディレクトリにPayrollSystemクラスを提案してください。

Task 4

設計ドキュメント #file:bonuscalculator-class.md に基づいて、javaディレクトリにBonusCalculatorクラスを提案してください。



Step 7

実践: Struts 1 からの移行

Task 0: ワークショップ用のリポジトリ

```
> git clone https://github.com/shinyay/code-to-code-ws-20250205-  
struts.git
```