JavaScript Todo CLI アプリケーション開発ガイド【Phase 2】

ゴール：JavaScript Todo CLI アプリケーションの作成

このガイドでは、JavaScriptの基礎から応用まで段階的に学習しながら、最終的にコマンドライン（CLI）で動作する多機能Todoアプリケーションを完成させます。

完成形のアプリケーション機能：

- タスクの追加・削除・完了
- 複数の表示方法（forEach, for, while）
- タスク検索機能
- 統計情報表示
- オブジェクト指向プログラミングの実践

一歩ずつ、着実にJavaScriptの知識を身につけていきましょう。

Section 0: プログラミングの世界への入門

ステップ0.1: プログラミングとは何か

プログラミングとは、コンピュータに対して「やってほしいこと」を正確に伝える作業です。料理のレシピのように、手順を順番に書いて、コンピュータがその通りに実行できるようにします。

プログラミングを身近な例で理解しましょう。

料理のレシピとプログラムの比較:

- 料理のレシピ: 「玉ねぎを切る → 炒める → 調味料を加える」
- プログラム: 「データを読み込む → 処理する → 結果を表示する」

どちらも「手順を正確に伝える」という点で同じです。

ステップ0.2: JavaScriptとは何か

JavaScriptは、Webサイトを動的にしたり、サーバーでプログラムを動かしたりできるプログラミング言語です。世界中で最も使われている言語の一つで、初心者にも学びやすい特徴があります。

JavaScriptの特徴:

- **書きやすい**: 文法が自然言語に近く、理解しやすい
- **多用途**: Webサイト、スマホアプリ、サーバーなど様々な用途に使える
- **豊富な情報**: インターネット上に学習資料がたくさんある
- **即座に実行**: 書いたコードをすぐに実行して結果を確認できる

なぜJavaScriptを学ぶのか:

1. **就職・転職に有利**: 求人数が多く、需要が高い
2. **副業にも活用**: Web制作やシステム開発の案件が豊富
3. **論理的思考力**: 問題を分解して解決する力が身につく
4. **創造性の発揮**: アイデアを形にする技術を習得できる

ステップ0.3: 今回作成するアプリケーションについて

CLIアプリケーションとは、Command Line Interface（コマンドライン インターフェース）の略で、文字だけでやり取りするプログラムです。マウスを使わず、キーボードだけで操作します。

今回作成するTodoアプリの機能:

- タスクの追加・削除・完了
- タスクの検索
- 統計情報の表示
- 複数の表示方法

なぜCLIアプリから始めるのか:

1. **本質に集中**: 見た目よりもプログラムの仕組みを理解できる
2. **基礎力向上**: プログラミングの基本概念をしっかり学べる
3. **実用性**: 実際に業務で使える実用的なツールを作成
4. **応用への土台**: Web版やスマホ版への発展が可能

ステップ0.4: 開発環境の理解

プログラミングには「開発環境」が必要です。これは、プログラムを書いて、実行して、確認するためのツール一式のことです。料理で言えば、包丁・まな板・コンロのような道具です。

必要なツール:

1. **Node.js**: JavaScriptを実行するエンジン
2. **VSCode**: プログラムを書くためのエディタ
3. **コマンドプロンプト**: プログラムを実行するためのツール

ステップ0.5: Node.jsのインストールと確認

Node.jsは、ブラウザの外でJavaScriptを実行できるようにするソフトウェアです。通常JavaScriptはWebサイトの中でしか動きませんが、Node.jsがあることで、パソコン上で直接JavaScriptプログラムを実行できます。

Node.jsの確認手順:

1. **Windowsの場合**: スタートメニューで「cmd」と検索 → 「コマンドプロンプト」をクリック **Macの場合**: LaunchpadでTerminalを検索 → 「ターミナル」をクリック

2. 黒い画面（コマンドプロンプト/ターミナル）が開いたら、以下を入力してEnterキーを押します：

```
node --version
```

3. **成功例**: `v18.17.0` のようなバージョン番号が表示される **失敗例**: `'node'は、内部コマンドまたは外部コマンドとして認識されていません`

Node.jsがインストールされていない場合:

1. https://nodejs.org/ にアクセス
2. 「LTS」版（推奨版）をダウンロード
3. インストーラーの指示に従ってインストール
4. 再度 `node --version` で確認

ステップ0.6: VSCodeのセットアップ

VSCode（Visual Studio Code）は、プログラムを書くための高機能なテキストエディタです。Microsoftが開発した無料のソフトで、世界中のプログラマーが使用しています。単なるメモ帳とは違い、プログラミングに特化した便利な機能がたくさん付いています。

VSCodeの便利な機能:

- **構文ハイライト**: コードが色分けされて見やすい
- **自動補完**: コードの候補を自動で表示
- **エラー検出**: 間違いを自動で指摘
- **ファイル管理**: プロジェクト全体を管理できる

VSCodeのインストール:

1. https://code.visualstudio.com/ にアクセス
2. 「Download for Windows/Mac」をクリック
3. インストーラーをダウンロードして実行
4. インストール完了後、VSCodeを起動

ステップ0.7: 作業ディレクトリの作成

プログラミングでは、関連するファイルをまとめて整理することが重要です。今回のプロジェクト専用のフォルダを作成し、そこですべての作業を行います。これにより、ファイルが散らばることを防ぎ、プロジェクト管理がしやすくなります。

フォルダ作成手順:

1. **デスクトップに新しいフォルダを作成**
   - デスクトップで右クリック → 「新規作成」→ 「フォルダー」
   - フォルダ名を「todo-cli-project」にします
2. **VSCodeでフォルダを開く**
   - VSCodeを起動
   - 「ファイル」メニュー → 「フォルダーを開く」
   - 先ほど作成した「todo-cli-project」フォルダを選択
3. **新しいファイルを作成**

- VSCodeの左側エクスプローラーで、フォルダ名の横の「新しいファイル」アイコンをクリック
- ファイル名を「todo.js」と入力してEnterキー

ステップ0.8: 初回動作確認

環境が正しくセットアップされているかを確認するため、簡単なプログラムを書いて実行してみます。この確認により、後のステップで問題が発生するリスクを事前に回避できます。

動作確認のためのテストコード:

VSCodeで作成した `todo.js` ファイルに以下を入力してください：

```
console.log(”Hello, JavaScript!”);
console.log(”Todo アプリケーション開発開始！”);
```

プログラムの実行:

1. **ファイルを保存**: Ctrl+S (Windows) または Cmd+S (Mac)
2. **コマンドプロンプト/ターミナルを開く**
3. **プロジェクトフォルダに移動**: 以下は一例、実際はあなたのフォルダの名前や場所に合わせる

```
cd Desktop/todo-cli-project
```

4. **プログラムを実行**:

```
node todo.js
```

期待される結果:

```
Hello, JavaScript!
Todo アプリケーション開発開始！
```

うまくいかない場合のトラブルシューティング:

エラーメッセージ	原因	解決方法
`node: command not found`	Node.jsがインストールされていない	ステップ0.5に戻ってNode.jsをインストール
`cannot find module`	ファイルパスが間違っている	`pwd`（Mac）または`dir`（Windows）で現在位置を確認
`syntax error`	コードに誤りがある	コードをもう一度確認し、正確に入力

ステップ0.9: プログラミングの基本的な考え方

プログラミングでは、大きな問題を小さな問題に分解して、一つずつ解決していくアプローチが重要です。これを「分割統治法」と呼びます。料理で複雑な料理を作る時も、「下ごしらえ」「調理」「盛り付け」に分けて考えるのと同じです。

今回作成するTodoアプリの分解:

1. **データの管理**: タスクの情報をどう保存するか
2. **機能の実装**: 追加・削除・表示などの操作
3. **ユーザーとの対話**: コマンドラインでの入出力
4. **エラー処理**: 問題が起きた時の対応

プログラミングの3つの基本構造:

1. **順次処理**: 上から下へ順番に実行
2. **条件分岐**: 「もしAならBをする、そうでなければCをする」
3. **繰り返し**: 「同じ処理を何度も実行する」

これらの組み合わせで、どんな複雑なプログラムも作ることができます。

✅ **準備完了チェック**

- ☐ Node.jsがインストールされ、バージョンが確認できる
- ☐ VSCodeがインストールされ、使用できる
- ☐ プロジェクトフォルダが作成され、VSCodeで開けている
- ☐ 簡単なJavaScriptプログラムが実行できる
- ☐ プログラミングの基本的な考え方を理解している

すべてにチェックが付いたら、次のセクションに進みましょう！

Section 1: JavaScript基礎概念の完全理解

ステップ1.1: 変数とは何か - データを保存する「箱」の概念

変数とは、データを保存する「名前付きの箱」です。現実世界でラベルを付けた箱に物を入れて整理するように、プログラムでもデータに名前を付けて管理します。JavaScriptでは、この箱を作るために `let`、`const`、`var` というキーワードを使います。

変数を日常生活で例えると:

- **現実の箱**: 「本の箱」「服の箱」「食器の箱」
- **プログラムの変数**: 「ユーザー名」「年齢」「タスクリスト」

なぜ変数が必要なのか:

1. **データの整理**: 散らばった情報をまとめて管理
2. **再利用**: 同じデータを何度も使用可能
3. **変更の容易性**: 一箇所変更すれば全体に反映
4. **可読性**: コードが理解しやすくなる

let, const, varの違い:

- **let**: 値を変更できる変数（一般的な箱）
- **const**: 値を変更できない定数（封印された箱）
- **var**: 古い書き方（現在は推奨されない）

最初の変数を宣言してみましょう：

```
___ userName = ”田中太郎”;
___ userAge = 25;
```

正解

```
let userName = "田中太郎";
let userAge = 25;
```

実際に確認してみよう: 上記のコードを `todo.js` に追加して実行してみてください。変数の値を `console.log()` で表示できます：

```
console.log("ユーザー名:", userName);
console.log("年齢:", userAge);
```

ステップ1.2: データ型の理解 - データの種類を知る

プログラムで扱うデータには種類があります。文字、数値、真偽値（はい・いいえ）など、それぞれ異なる特徴を持ちます。これを「データ型」と呼びます。適切なデータ型を使うことで、バグを防ぎ、効率的なプログラムを作成できます。

主要なデータ型:

1. **文字列 (String)**: 文字や文章

```
let message = "こんにちは";  // ダブルクォート
let name = 'JavaScript';    // シングルクォート
```

2. **数値 (Number)**: 数字

```
let age = 25;        // 整数
let price = 99.99;   // 小数
```

3. **真偽値 (Boolean)**: trueまたはfalse

```
let isCompleted = true;   // 完了している
let isVisible = false;    // 表示されていない
```

4. **配列 (Array)**: 複数のデータのリスト

```
let fruits = ["りんご", "バナナ", "オレンジ"];
```

5. **オブジェクト (Object)**: 関連するデータのまとまり

```
let person = {
    name: "田中",
    age: 25,
    isStudent: true
};
```

データ型を確認する方法:

```
let testData = "Hello";
console.log("データ型:", ___ testData);
```

正解

```
let testData = "Hello";
console.log("データ型:", typeof testData);
```

ステップ1.3: 配列の概念と必要性 - 複数のデータを整理する

配列は、同じ種類の複数のデータを順序立てて保存するデータ構造です。本棚に本を順番に並べるように、データを0番、1番、2番...という番号（インデックス）で管理します。Todoアプリでは、複数のタスクを配列で管理します。

なぜ配列が必要なのか:

悪い例（配列を使わない場合）:

```
let task1 = "買い物";
let task2 = "掃除";
let task3 = "勉強";
// タスクが100個あったら、task1, task2, ... task100まで書く必要がある
```

良い例（配列を使う場合）:

```
let tasks = ["買い物", "掃除", "勉強"];
// いくつタスクが増えても配列一つで管理できる
```

配列の基本操作:

```
// 空の配列を作成
let tasks = [];

// 要素を追加
tasks.___(”新しいタスク”);

// 要素の数を確認
console.log(”タスク数:”, tasks.___);

// 特定の要素にアクセス（最初は0番）
console.log(”最初のタスク:”, tasks[0]);
```

正解

```
// 空の配列を作成
let tasks = [];

// 要素を追加
tasks.push(”新しいタスク”);

// 要素の数を確認
console.log(”タスク数:”, tasks.length);

// 特定の要素にアクセス（最初は0番）
console.log(”最初のタスク:”, tasks[0]);
```

配列のインデックスの重要な特徴:

- インデックスは0から始まる（1番目の要素は[0]）
- 存在しないインデックスにアクセスすると`undefined`が返される

ステップ1.4: オブジェクトの概念 - 関連する情報をまとめる

オブジェクトは、関連する情報をプロパティ（属性）としてまとめて管理するデータ構造です。人物カードに「名前」「年齢」「職業」などの情報をまとめて書くように、プログラムでも関連するデータを一つのオブジェクトにまとめます。

オブジェクトを現実世界で例えると:

人物の情報:

- 名前: 田中太郎
- 年齢: 25歳
- 職業: エンジニア
- 趣味: 読書

Todoタスクの情報:

- タイトル: 買い物
- 説明: 牛乳と卵を購入
- 優先度: 高
- 完了状態: 未完了

オブジェクトの作成方法:

```
let person = {
    name: ”田中太郎”,      // プロパティ名: 値
    age: ___,             // 数値
    isStudent: ___,       // 真偽値
    hobbies: [”読書”, ”映画鑑賞”]  // 配列
};
```

正解

```
let person = {
    name: ”田中太郎”,
    age: 25,
    isStudent: false,
    hobbies: [”読書”, ”映画鑑賞”]
};
```

オブジェクトのプロパティにアクセスする方法:

```
// ドット記法
console.log(person.name);     // ”田中太郎”
console.log(person.age);      // 25

// ブラケット記法
console.log(person[”name”]);  // ”田中太郎”
```

ステップ1.5: 関数の概念と重要性 - 処理をまとめて再利用する

関数は、特定の処理をまとめて名前を付けたものです。料理のレシピのように、一度書いておけば何度でも使えます。「材料（引数）を渡すと、料理（戻り値）が完成する」という仕組みです。関数を使うことで、同じコードを何度も書く必要がなくなり、保守性が向上します。

関数を料理のレシピで例えると:

カレーのレシピ（関数）:

- 材料（引数）: 肉、野菜、カレー粉
- 手順（処理）: 切る → 炒める → 煮込む
- 完成品（戻り値）: カレー

プログラムの関数:

- 材料（引数）: タイトル、説明、優先度
- 手順（処理）: データを整理してオブジェクトを作成
- 完成品（戻り値）: タスクオブジェクト

関数の基本構文:

```
___ addNumbers(a, b) {
    let result = a + b;
    ___ result;
}

// 関数の使用
let sum = addNumbers(5, 3);
console.log(sum); // 8
```

正解

```
function addNumbers(a, b) {
    let result = a + b;
    return result;
}

// 関数の使用
```

```
let sum = addNumbers(5, 3);
console.log(sum); // 8
```

関数を使う利点:

1. **再利用性**: 一度書けば何度でも使える
2. **保守性**: 修正が一箇所で済む
3. **可読性**: 処理の内容が名前で分かる
4. **テスト性**: 個別に動作確認できる

ステップ1.6: Todoアプリ用のタスク作成関数

実際にTodoアプリで使用するタスク作成関数を実装しましょう。

```
___ createTask(title, desc, priority, important) {
    return {
        title: title || ”無題”,
        desc: desc ?? ”説明なし”,
        priority: Number(priority),
        important: Boolean(important),
        completed: false,
        created: new Date()
    };
}
```

正解

```
function createTask(title, desc, priority, important) {
    return {
        title: title || ”無題”,
        desc: desc ?? ”説明なし”,
        priority: Number(priority),
        important: Boolean(important),
        completed: false,
        created: new Date()
    };
}
```

ステップ1.7: 論理演算子とデフォルト値 - 安全なプログラムを作る

プログラムでは、予期しないデータ（空文字、null、undefined）が渡される可能性があります。論理演算子を使ってデフォルト値を設定することで、エラーを防ぎ、安全なプログラムを作成できます。

論理演算子の種類と使い分け:

1. **|| (OR演算子)**: 左側がfalsyなら右側を使用
   - falsy値: `false`, `0`, `""`, `null`, `undefined`, `NaN`
2. **?? (Null合体演算子)**: 左側がnullまたはundefinedなら右側を使用

実際の例で違いを確認:

```
// || の場合
let title1 = "" || "無題";         // 結果: ___
let title2 = null || "無題";       // 結果: ___
let title3 = "買い物" || "無題";   // 結果: ___

// ?? の場合
let desc1 = "" ?? "説明なし";      // 結果: ___
let desc2 = null ?? "説明なし";    // 結果: ___
let desc3 = "重要な会議" ?? "説明なし"; // 結果: ___
```

正解

```
// || の場合
let title1 = "" || "無題";         // 結果: "無題"
let title2 = null || "無題";       // 結果: "無題"
let title3 = "買い物" || "無題";   // 結果: "買い物"

// ?? の場合
let desc1 = "" ?? "説明なし";      // 結果: ""
let desc2 = null ?? "説明なし";    // 結果: "説明なし"
let desc3 = "重要な会議" ?? "説明なし"; // 結果: "重要な会議"
```

ステップ1.8: 型変換の重要性と実践

ユーザーからの入力は通常文字列として受け取られますが、プログラム内では適切なデータ型に変換する必要があります。型変換を行うことで、意図しない動作

やエラーを防げます。

型変換が必要な理由:

```
// 問題のある例（型変換なし）
let age1 = "25";
let age2 = "30";
console.log(age1 + age2); // "2530"（文字列として結合される）

// 正しい例（型変換あり）
let age3 = Number("25");
let age4 = Number("30");
console.log(age3 + age4); // 55（数値として計算される）
```

型変換関数の使用:

```
// 文字列を数値に変換
let priority = ___("3");
let importance = ___("true");

console.log("優先度:", priority, "型:", typeof priority);
console.log("重要度:", importance, "型:", typeof importance);
```

正解

```
// 文字列を数値に変換
let priority = Number("3");
let importance = Boolean("true");

console.log("優先度:", priority, "型:", typeof priority);
console.log("重要度:", importance, "型:", typeof importance);
```

型変換の注意点:

- `Number("abc")` → `NaN` (Not a Number)
- `Boolean("")` → `false`
- `Boolean("false")` → `true` (文字列"false"は真値)

✅ **Section 1 理解度チェック**

- ☐ 変数の概念と用途を理解している
- ☐ 基本的なデータ型を区別できる
- ☐ 配列の基本操作ができる
- ☐ オブジェクトの作成とアクセスができる
- ☐ 関数の定義と呼び出しができる
- ☐ 論理演算子の使い分けができる
- ☐ 型変換の重要性を理解している

次のセクションに進む前に、実際にコードを書いて動作確認をしましょう！

Section 2: 配列操作メソッドの習得

ステップ2.1: forEach メソッドの基本理解

forEachは、配列の各要素に対して指定した関数を順番に実行するメソッドです。forループとは異なり、インデックスの管理が自動化されるため、エラーが発生しにくく、コードが読みやすくなります。

forEachの基本構文：

```
配列名.forEach(function(要素, インデックス) {
    // 各要素に対する処理
});
```

従来のforループとの比較：

```
// forループ（手動管理）
for (let i = 0; i < tasks.length; i++) {
    console.log(tasks[i].title);
}

// forEach（自動管理）
tasks.forEach(function(task, index) {
    console.log(task.title);
});
```

下のコードを完成させてください：

```
let fruits = ["りんご", "バナナ", "オレンジ"];
fruits.___(___(fruit, index) {
    console.log(`${index}: ${fruit}`);
});
```

正解

```
let fruits = ["りんご", "バナナ", "オレンジ"];
fruits.forEach(function(fruit, index) {
    console.log(`${index}: ${fruit}`);
});
```

ステップ2.2: アロー関数の基本理解

**アロー関数は、function キーワードを使わずに関数を簡潔に書ける構文です。
ES6で導入され、特にコールバック関数として広く使用されます。基本的な変換ルールを覚えることで、既存のfunction記法をアロー関数に変換できます。**

アロー関数の変換ルール：

1. `function` キーワードを削除
2. 引数の後に `=>` を記述
3. 処理内容は変更しない

変換の実例：

```
// function記法
function(task, index) {
    console.log(task.title);
}

// アロー関数に変換
(task, index) => {
    console.log(task.title);
}
```

下のfunction記法をアロー関数に変換してください：

```
// function記法
numbers.forEach(function(num) {
    console.log(num * 2);
});

// アロー関数版
numbers.forEach(___);
```

正解

```
// function記法
numbers.forEach(function(num) {
    console.log(num * 2);
});

// アロー関数版
numbers.forEach((num) => {
    console.log(num * 2);
});
```

ステップ2.3: テンプレートリテラルの基本理解

テンプレートリテラルはES6で導入された文字列処理機能です。従来の文字列連結(`+`演算子)と比較して：

1. **可読性向上**：改行を直接表現可能
2. **安全な変数展開**：自動型変換による意図しない連結を防止
3. **式の埋め込み**：`${}`内で計算や関数呼び出しが可能

実務での主な使用例：

- 動的HTML生成
- 複雑なエラーメッセージの構築
- 国際化対応の多言語テンプレート

従来の方法との比較：

```
// 従来の文字列連結
let message = "ユーザー名: " + userName + ", 年齢: " + age;

// テンプレートリテラル
let message = `ユーザー名: ${userName}, 年齢: ${age}`;
```

下の文字列連結をテンプレートリテラルに変換してください：

```
let itemNumber = 3;
let itemName = "ペン";
let price = 150;

// 従来の文字列連結
let info = itemNumber + "番: " + itemName + " (" + price + "円)";

// テンプレートリテラル版
let info2 = ___;
```

正解

```
let itemNumber = 3;
let itemName = "ペン";
let price = 150;

// 従来の文字列連結
let info = itemNumber + "番: " + itemName + " (" + price + "円)";

// テンプレートリテラル版
let info2 = `${itemNumber}番: ${itemName} (${price}円)`;
```

ステップ2.4: 三項演算子の理解

三項演算子（`? :`）は、条件によって異なる値を返すための簡潔な書き方です。
if文を短く書くことができ、テンプレートリテラルの中でよく使用されます。

三項演算子の基本構文：

```
条件 ? 真の場合の値 : 偽の場合の値
```

if文との比較：

```
// if文を使った場合
let mark;
if (task.completed) {
    mark = "x";
} else {
    mark = " ";
}

// 三項演算子を使った場合
let mark = task.completed ? "x" : " ";
```

下の穴埋めを完成させてください：

```
let isCompleted = true;
let status = isCompleted ? ___ : ___;
console.log(status); // "完了" と表示される
```

正解

```
let isCompleted = true;
let status = isCompleted ? "完了" : "未完了";
console.log(status); // "完了" と表示される
```

ステップ2.5: テンプレートリテラルと三項演算子の組み合わせ

これまで学習した知識を組み合わせて、より複雑な表示を作成しましょう。

```
let i = 0;
let task = {
    title: "買い物",
    completed: false
};

// テンプレートリテラルと三項演算子を組み合わせ
console.log(`${i + 1}: [${task.completed ? ___ : ___}] ${task.title}`);
```

正解

```
let i = 0;
let task = {
    title: "買い物",
    completed: false
};

// テンプレートリテラルと三項演算子を組み合わせ
console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
```

ステップ2.6: 条件分岐（if文）の基本復習

`if` 文は、条件によって処理を分岐させるための基本的な制御構文です。配列が空の場合の処理など、エラーを防ぐための重要な仕組みです。

if文の基本構文：

```
if (条件) {
    // 条件が真の場合の処理
}
```

配列の長さをチェックする例：

```
if (tasks.length === 0) {
    console.log("タスクがありません");
}
```

下の穴埋めを完成させてください：

```
if (tasks.___ === 0) {
    console.log("タスクがありません");
}
```

正解

```
if (tasks.length === 0) {
    console.log("タスクがありません");
}
```

ステップ2.7: return文の理解

`return`文は、関数の実行を終了し、呼び出し元に値を返すための構文です。配列が空の場合など、これ以上処理を続ける必要がない時に使用します。

return文の役割：

```
function checkTasks() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return; // ここで関数を終了（以降の処理を実行しない）
    }

    console.log("タスクがあります"); // 上でreturnされた場合、この行は実行されない
}
```

下の穴埋めを完成させてください：

```
function showTaskCount() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        ___; // 関数を終了
    }

    console.log("タスク数:", tasks.length);
}
```

正解

```
function showTaskCount() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return; // 関数を終了
    }
```

```
    console.log("タスク数:", tasks.length);
}
```

ステップ2.8: 学習内容の統合演習

これまで学習した4つの要素を組み合わせて、完全なタスク表示関数を作成します。

プログラミングでは、個別に学習した概念を組み合わせて実用的な機能を実装します。forEach、アロー関数、テンプレートリテラル、三項演算子を統合することで、読みやすく効率的なコードが書けます。

統合する要素の確認：

1. forEach メソッド（配列の反復処理）
2. アロー関数（簡潔な関数記法）
3. テンプレートリテラル（文字列への変数埋め込み）
4. 三項演算子（条件による値の選択）

下の関数を完成させてください：

```
function listTasksForEach() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return;
    }

    // 4つの要素を全て使って実装
    tasks.___((task, i) => {
        console.log(___);
    });
}
```

表示形式： `1: [x] 買い物` （完了済みはx、未完了は空白）

正解

```
function listTasksForEach() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return;
```

```
    }

    // 4つの要素を全て使って実装
    tasks.forEach((task, i) => {
        console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
    });
}
```

ステップ2.9: for ループの基本理解

`for` ループは、初期値、条件、増減処理を明示的に指定する繰り返し構文です。
配列のインデックスを直接制御したい場合に使用します。forEachと同じ結果を得ることができますが、より詳細な制御が可能です。

for ループの基本構文：

```
for (初期値; 条件; 増減処理) {
    // 繰り返したい処理
}
```

配列を使ったfor ループの例：

```
for (let i = 0; i < tasks.length; i++) {
    const task = tasks[i];
    console.log(task.title);
}
```

下の穴埋めを完成させてください：

```
for (let i = ___; i < tasks.___; ___) {
    const task = tasks[i];
    console.log(`${i + 1}: ${task.title}`);
}
```

正解

```
for (let i = 0; i < tasks.length; i++) {
    const task = tasks[i];
    console.log(`${i + 1}: ${task.title}`);
}
```

ステップ2.10: for ループを使った完全な関数の作成

同じ表示処理を`for`ループで実装しましょう。

```
function listTasksFor() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return;
    }

    for (let i = ___; i < tasks.___; ___) {
        const task = tasks[i];
        console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
    }
}
```

正解

```
function listTasksFor() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return;
    }

    for (let i = 0; i < tasks.length; i++) {
        const task = tasks[i];
        console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
    }
}
```

ステップ2.11: while ループの基本理解

`while`ループは、指定した条件がtrueの間、処理を繰り返します。条件が最初からfalseの場合、一度も実行されません。手動でカウンターを管理する必要があります。

while ループの基本構文：

```
while (条件) {
    // 繰り返したい処理
}
```

配列を使ったwhile ループの例：

```
let i = 0;
while (i < tasks.length) {
    const task = tasks[i];
    console.log(task.title);
    i++; // カウンターを手動で増加
}
```

下の穴埋めを完成させてください：

```
let i = ___;
___ (i < tasks.length) {
    const task = tasks[i];
    console.log(`${i + 1}: ${task.title}`);
    ___;
}
```

正解

```
let i = 0;
while (i < tasks.length) {
    const task = tasks[i];
    console.log(`${i + 1}: ${task.title}`);
    i++;
}
```

ステップ2.12: while ループを使った完全な関数の作成

同じ表示処理を `while` ループで実装しましょう。

```
function listTasksWhile() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return;
    }

    let i = ___;
    ___ (i < tasks.length) {
        const task = tasks[i];
        console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
        ___;
    }
}
```

正解

```
function listTasksWhile() {
    if (tasks.length === 0) {
        console.log("タスクがありません");
        return;
    }

    let i = 0;
    while (i < tasks.length) {
        const task = tasks[i];
        console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
        i++;
    }
}
```

ステップ2.13: includes メソッドの基本理解

`includes`は、文字列に特定の文字列が含まれているかを判定するメソッドです。大文字小文字を区別し、完全一致ではなく部分一致で検索します。検索機能の実装において基礎となるメソッドです。

includesの基本例:

```
let title = "買い物リスト";

// 部分一致の検索
console.log(title.includes("買い"));    // true
```

```
console.log(title.includes("物"));      // true
console.log(title.includes("買い物"));   // true
console.log(title.includes("掃除"));    // false
```

下の穴埋めを完成させてください：

```
let taskTitle = "重要な会議の準備";
console.log(taskTitle.___("会議"));  // true が表示される
```

正解

```
let taskTitle = "重要な会議の準備";
console.log(taskTitle.includes("会議"));  // true が表示される
```

ステップ2.14: filter メソッドの基本理解

filterは、配列から条件に合う要素のみを抽出して新しい配列を作成するメソッドです。元の配列は変更されず、指定した条件を満たす要素だけで構成された配列が返されます。データの絞り込みや検索機能の実装において基礎となるメソッドです。

filterの基本構文：

```
新しい配列 = 元の配列.filter(要素 => 条件式);
```

数値の絞り込み例：

```
let numbers = [1, 2, 3, 4, 5];
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4]
console.log(numbers);     // [1, 2, 3, 4, 5] (元の配列は変更されない)
```

下の条件に合う要素を抽出してください：

```
let ages = [15, 22, 17, 25, 19];
// 20歳以上の年齢のみを抽出
let adults = ages.___(age => age ___ 20);
console.log(adults); // [22, 25] と表示される
```

正解

```
let ages = [15, 22, 17, 25, 19];
// 20歳以上の年齢のみを抽出
let adults = ages.filter(age => age >= 20);
console.log(adults); // [22, 25] と表示される
```

ステップ2.15: filter と includes を組み合わせた検索機能

filterとincludesを組み合わせることで、強力な検索機能を実装できます。

```
// タスクタイトルに特定の文字列が含まれるタスクを検索
let searchWord = "買い";
let foundTasks = tasks.___(task => task.title.___(searchWord));
```

下の穴埋めを完成させてください：

```
let searchWord = "買い";
let foundTasks = tasks.___(task => task.title.___(searchWord));
```

正解

```
let searchWord = "買い";
let foundTasks = tasks.filter(task => task.title.includes(searchWord));
```

ステップ2.16: 検索結果の処理と表示

検索結果が存在するかどうかを確認し、適切に表示する処理を学習しましょう。

`length`プロパティを使用して配列の要素数を確認することで、検索結果が存在するかを判定できます。結果が0の場合は「見つからない」メッセージを表示し、1以上の場合は結果を表示します。

検索機能の完全な実装を作成してみましょう：

```
function searchTasks(searchWord) {
    const foundTasks = tasks.filter(task => task.title.includes(searchWord));

    if (foundTasks.___ > 0) {
        foundTasks.forEach((task, i) => {
            console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
        });
    } else {
        console.log("該当するタスクが見つかりません");
    }
}
```

正解

```
function searchTasks(searchWord) {
    const foundTasks = tasks.filter(task => task.title.includes(searchWord));

    if (foundTasks.length > 0) {
        foundTasks.forEach((task, i) => {
            console.log(`${i + 1}: [${task.completed ? "x" : " "}] ${task.title}`);
        });
    } else {
        console.log("該当するタスクが見つかりません");
    }
}
```

ステップ2.17：否定演算子（!）の基本理解

否定演算子（`!`）は、真偽値を反転させる演算子です。`true`を`false`に、`false`を`true`に変換します。「〜でない」という条件を表現する際に使用します。

否定演算子の基本例：

```
let isCompleted = true;
let isNotCompleted = !isCompleted;  // false

let isEmpty = false;
let isNotEmpty = !isEmpty;  // true
```

タスクでの使用例:

```
// 完了していないタスク
task.completed      // 完了している場合true
!task.completed     // 完了していない場合true
```

下の穴埋めを完成させてください：

```
let isFinished = false;
let isNotFinished = ___isFinished;
console.log(isNotFinished); // true が表示される
```

正解

```
let isFinished = false;
let isNotFinished = !isFinished;
console.log(isNotFinished); // true が表示される
```

ステップ2.18: every メソッドの基本理解

`every`は、配列のすべての要素が指定した条件を満たすかどうかを判定するメソッドです。一つでも条件を満たさない要素があれば`false`を返し、すべての要素が条件を満たす場合のみ`true`を返します。「全件チェック」の処理において重要なメソッドです。

everyの基本構文:

```
配列.every(要素 => 条件)
```

everyの動作例:

```
let numbers = [2, 4, 6, 8];
let allEven = numbers.every(num => num % 2 === 0);  // すべて偶数？
console.log(allEven); // true

let mixed = [2, 4, 5, 8];
let allEvenMixed = mixed.every(num => num % 2 === 0);  // すべて偶数？
console.log(allEvenMixed); // false (5が奇数のため)
```

下の穴埋めを完成させてください：

```
let tasks = [
    {completed: true},
    {completed: true},
    {completed: true}
];

let allTasksCompleted = tasks.___(task => task.completed);
console.log("すべて完了:", allTasksCompleted); // true が表示される
```

正解

```
let tasks = [
    {completed: true},
    {completed: true},
    {completed: true}
];

let allTasksCompleted = tasks.every(task => task.completed);
console.log("すべて完了:", allTasksCompleted); // true が表示される
```

ステップ2.19: some メソッドの基本理解

`some`は、配列の要素の中で少なくとも一つが指定した条件を満たすかどうかを判定するメソッドです。一つでも条件を満たす要素があれば`true`を返し、すべての要素が条件を満たさない場合のみ`false`を返します。「存在チェック」の処理において重要なメソッドです。

someの基本構文:

```
配列.some(要素 => 条件)
```

someの動作例:

```
let numbers = [1, 3, 5, 7];
let hasEven = numbers.some(num => num % 2 === 0);  // 偶数が存在する？
console.log(hasEven); // false

let mixed = [1, 3, 4, 7];
let hasEvenMixed = mixed.some(num => num % 2 === 0);  // 偶数が存在する？
console.log(hasEvenMixed); // true (4が偶数のため)
```

下の穴埋めを完成させてください：

```
let tasks = [
    {completed: true},
    {completed: false},
    {completed: true}
];

// 未完了のタスクが存在するかチェック
let hasIncompleteTask = tasks.___(task => ___task.completed);
console.log("未完了あり:", hasIncompleteTask); // true が表示される
```

正解

```
let tasks = [
    {completed: true},
    {completed: false},
    {completed: true}
];

// 未完了のタスクが存在するかチェック
let hasIncompleteTask = tasks.some(task => !task.completed);
console.log("未完了あり:", hasIncompleteTask); // true が表示される
```

ステップ2.20: everyとsomeの違いと使い分け

`every`と`some`は対になるメソッドです。everyは「全部」、someは「一部」の条件チェックに使用します。否定演算子と組み合わせることで、様々な条件判定が可能になります。

everyとsomeの違い：

- `every`：**すべて**が条件を満たすか → 全員が合格しているか？
- `some`：**少なくとも一つ**が条件を満たすか → 不合格者がいるか？

実際の使用例：

```
// 重要でないタスクを見つける
let notImportantTasks = tasks.filter(task => ___task.important);
```

正解

```
// 重要でないタスクを見つける
let notImportantTasks = tasks.filter(task => !task.important);
```

ステップ2.21：統計機能の完全実装

これまで学習した知識を組み合わせて、完全な統計表示機能を作成しましょう。

必要な知識の確認：

1. ✅ filter メソッド（完了済みタスクの抽出）
2. ✅ every メソッド（全タスク完了チェック）
3. ✅ some メソッド（未完了タスク存在チェック）
4. ✅ 否定演算子（!）
5. ✅ テンプレートリテラル

```
function showStatistics() {
    const total = tasks.length;
    const completed = tasks.___ (task => task.completed).length;
    const allCompleted = tasks.___(task => task.completed);
    const hasIncomplete = tasks.___(task => ___task.completed);
```

```
    console.log(`総タスク数: ${total}`);
    console.log(`完了済み: ${completed}`);
    console.log(`未完了: ${total - completed}`);
    console.log(`すべて完了: ${allCompleted ? "はい" : "いいえ"}`);
    console.log(`未完了あり: ${hasIncomplete ? "はい" : "いいえ"}`);
}
```

正解

```
function showStatistics() {
    const total = tasks.length;
    const completed = tasks.filter(task => task.completed).length;
    const allCompleted = tasks.every(task => task.completed);
    const hasIncomplete = tasks.some(task => !task.completed);

    console.log(`総タスク数: ${total}`);
    console.log(`完了済み: ${completed}`);
    console.log(`未完了: ${total - completed}`);
    console.log(`すべて完了: ${allCompleted ? "はい" : "いいえ"}`);
    console.log(`未完了あり: ${hasIncomplete ? "はい" : "いいえ"}`);
}
```

ステップ2.22: mapメソッドの基本理解

mapは、配列の各要素に変換処理を適用して新しい配列を作成するメソッドです。元の配列は変更されず、変換結果として新しい配列が返されます。データの表示形式変更やデータ加工において重要な役割を果たします。

mapの基本構文：

```
新しい配列 = 元の配列.map(要素 => 変換処理);
```

シンプルな変換例：

```
let numbers = [1, 2, 3];
let doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
console.log(numbers); // [1, 2, 3] (元の配列は変更されない)
```

下の配列変換を完成させてください：

```
let prices = [100, 200, 300];
// 全ての価格に税率1.1を掛けた新しい配列を作成
let taxIncluded = prices.___(price => price * ___);
console.log(taxIncluded); // [110, 220, 330] と表示される
```

正解

```
let prices = [100, 200, 300];
// 全ての価格に税率1.1を掛けた新しい配列を作成
let taxIncluded = prices.map(price => price * 1.1);
console.log(taxIncluded); // [110, 220, 330] と表示される
```

ステップ2.23: 配列メソッドの使い分けまとめ

JavaScript の配列メソッドは、それぞれ異なる目的を持ちます。適切な場面で適切なメソッドを選択することで、効率的で読みやすいコードを書くことができます。

各メソッドの特徴と使用場面:

メソッド	用途	戻り値	使用例
`forEach`	全要素の処理	なし	全タスクの表示
`map`	要素の変換	新しい配列	番号付きリスト作成
`filter`	条件による絞り込み	新しい配列	検索、未完了タスク抽出
`every`	全要素が条件を満たすか	true/false	全タスク完了チェック
`some`	一つでも条件を満たすか	true/false	未完了タスク存在チェック

実際の使用例:

```
// すべての要素を処理
tasks.forEach(task => console.log(task.title));

// 要素を変換
let numberedTasks = tasks.map((task, i) => `${i + 1}. ${task.title}`);

// 条件で絞り込み
let incompleteTasks = tasks.filter(task => !task.completed);

// 全部チェック
let allDone = tasks.every(task => task.completed);

// 存在チェック
let hasUrgent = tasks.some(task => task.priority > 3);
```

✅ **Section 2 理解度チェック**

- ☐ forEach メソッドを使った配列の処理ができる
- ☐ アロー関数の基本的な書き方を理解している
- ☐ テンプレートリテラルを使った文字列作成ができる
- ☐ 三項演算子を使った条件分岐ができる
- ☐ for ループと while ループの違いを理解している
- ☐ map メソッドを使った配列の変換ができる
- ☐ filter メソッドを使った配列の絞り込みができる
- ☐ includes メソッドを使った文字列検索ができる
- ☐ every と some メソッドの使い分けができる
- ☐ 否定演算子を適切に使用できる

次のセクションに進む前に、実際にコードを書いて動作確認をしましょう！

Section 3: 比較演算子と条件分岐の深い理解

ステップ3.1: 等価演算子の基本概念

JavaScriptには2種類の等価演算子があります。`==`（等価演算子）は型変換を行って比較し、`===`（厳密等価演算子）は型変換を行わずに比較します。この違

いを理解することは、バグを防ぎ、予期しない動作を避けるために非常に重要です。

なぜ2種類の等価演算子があるのか：

JavaScriptは「動的型付け言語」のため、同じ値でも異なるデータ型で表現される場合があります。

型変換の例：

```
let numberValue = 5;       // 数値の5
let stringValue = "5";     // 文字列の"5"

// 見た目は同じ「5」だが、データ型が異なる
console.log(typeof numberValue); // "number"
console.log(typeof stringValue); // "string"
```

ステップ3.2: ==（等価演算子）の動作理解

`==` は比較時に自動的に型変換を行います。異なるデータ型同士でも、値が「同じと見なせる」場合はtrueを返します。一見便利ですが、予期しない結果を生むことがあるため注意が必要です。

== による型変換の例：

```
console.log(5 == "5");        // true（文字列"5"が数値5に変換される）
console.log(true == 1);       // true（trueが1に変換される）
console.log(false == 0);      // true（falseが0に変換される）
console.log(null == undefined); // true（特殊ルール）
```

下の結果を予想してください：

```
console.log(0 == "");         // 結果: ___
console.log(0 == false);      // 結果: ___
```

正解

```
console.log(0 == "");         // 結果: true
console.log(0 == false);      // 結果: true
```

ステップ3.3: ===（厳密等価演算子）の動作理解

`===` は型変換を一切行わず、値とデータ型の両方が完全に一致する場合のみtrueを返します。より予測可能で安全な比較を行うため、現代のJavaScript開発では `===` の使用が強く推奨されています。

=== による厳密な比較の例：

```
console.log(5 === "5");         // false（数値と文字列は異なる型）
console.log(true === 1);        // false（真偽値と数値は異なる型）
console.log(false === 0);       // false（真偽値と数値は異なる型）
console.log(null === undefined); // false（異なる型）
```

下の結果を予想してください：

```
console.log(5 === 5);           // 結果: ___
console.log("hello" === "hello"); // 結果: ___
console.log(true === true);     // 結果: ___
```

正解

```
console.log(5 === 5);           // 結果: true
console.log("hello" === "hello"); // 結果: true
console.log(true === true);     // 結果: true
```

ステップ3.4: 不等価演算子の理解

等価演算子と同様に、不等価演算子にも2種類あります。 `!=` は型変換を行い、`!==` は型変換を行いません。「等しくない」ことを判定する際に使用します。

不等価演算子の例：

```
// != （型変換あり）
console.log(5 != "5");          // false（変換後に等しいため）
console.log(5 != "6");          // true（変換後も等しくないため）

// !== （型変換なし）
console.log(5 !== "5");         // true（型が異なるため）
console.log(5 !== 5);           // false（完全に同じため）
```

下の穴埋めを完成させてください：

```
let userAge = 25;
let inputAge = "25";

// 値は同じだが型が違うかチェック
if (userAge ___ inputAge) {
    console.log("型が異なります");
}
```

正解

```
let userAge = 25;
let inputAge = "25";

// 値は同じだが型が違うかチェック
if (userAge !== inputAge) {
    console.log("型が異なります");
}
```

ステップ3.5: 実践例：統計機能での比較演算子活用

実際のTodoアプリで比較演算子の違いを確認する機能を実装しましょう。

```
function showDetailedStats() {
    const total = tasks.length;
    const completed = tasks.filter(task => task.completed).length;

    // 等価演算子の比較（型変換あり）
    const isEqual = total ___ completed;

    // 厳密等価演算子の比較（型変換なし）
    const isStrictEqual = total ___ completed;
```

```
    console.log(`総タスク数: ${total} (型: ${typeof total})`);
    console.log(`完了数: ${completed} (型: ${typeof completed})`);
    console.log(`==での比較: ${isEqual}`);
    console.log(`===での比較: ${isStrictEqual}`);

    // 実際には同じ結果になる（どちらもnumber型のため）
    console.log("この場合、どちらも同じ結果になります");
}
```

正解

```
function showDetailedStats() {
    const total = tasks.length;
    const completed = tasks.filter(task => task.completed).length;

    // 等価演算子の比較（型変換あり）
    const isEqual = total == completed;

    // 厳密等価演算子の比較（型変換なし）
    const isStrictEqual = total === completed;

    console.log(`総タスク数: ${total} (型: ${typeof total})`);
    console.log(`完了数: ${completed} (型: ${typeof completed})`);
    console.log(`==での比較: ${isEqual}`);
    console.log(`===での比較: ${isStrictEqual}`);

    // 実際には同じ結果になる（どちらもnumber型のため）
    console.log("この場合、どちらも同じ結果になります");
}
```

ステップ3.6: ベストプラクティス：常に===を使用する理由

現代のJavaScript開発では、意図的な理由がない限り `===` と `!==` を使用することが強く推奨されています。これにより、型変換による予期しない動作を防ぎ、コードの意図がより明確になります。

==による問題の例：

```
// 問題のあるコード
if (userInput == 0) {
    console.log("入力が0です");
}
```

```
// 問題：userInputが空文字””の場合もtrueになってしまう

// 改善されたコード
if (userInput === 0) {
    console.log(”入力が0です”);
}
// 改善：厳密に数値の0のみをチェック
```

推奨される書き方：

- `===` と `!==` を基本的に使用
- `==` と `!=` は特別な理由がある場合のみ
- リント（コード品質チェックツール）でも `===` の使用が推奨される

Section 4: エラーハンドリング - 安全なプログラムの作成

ステップ4.1: エラーとは何か

エラーとは、プログラムが正常に実行できない状況のことです。エラーが発生すると、プログラムはその時点で停止し、後続の処理は実行されません。エラーを適切に処理することで、プログラムの安定性を保ち、ユーザーに分かりやすいメッセージを提供できます。

エラーが発生する典型的な状況：

例1: 存在しない配列要素へのアクセス

```
let tasks = [”買い物”, ”掃除”];
console.log(tasks[10]); // undefined（エラーではない）
console.log(tasks[10].title); // TypeError（エラー発生）
```

例2: nullやundefinedのプロパティアクセス

```
let task = null;
console.log(task.title); // TypeError: Cannot read property 'title' of null
```

下の状況でエラーが発生するコードを特定してください：

```
let data = undefined;
let result1 = data + 5;        // A: 計算実行
let result2 = data.length;     // B: プロパティアクセス
let result3 = data[0];         // C: 配列アクセス
```

どの行でエラーが発生しますか？ ___

正解

どの行でエラーが発生しますか？ **B と C**

解説：

- A: `undefined + 5` = `NaN`（エラーではない）
- B: `undefined.length` = TypeError（エラー）
- C: `undefined[0]` = TypeError（エラー）

ステップ4.2: エラーが起きるとどうなるか

JavaScriptでエラーが発生すると、その時点でプログラムの実行が停止します。エラーの後に書かれたコードは実行されません。この動作により、ユーザーにとって不親切な状況や、データの破損などが起こる可能性があります。

エラー発生時の動作例：

```
console.log("プログラム開始");

// ここでエラーが発生
let invalidArray = null;
console.log(invalidArray.length); // TypeError が発生

console.log("この行は実行されない"); // エラーで停止するため
```

実行結果：

```
プログラム開始
TypeError: Cannot read property 'length' of null
// "この行は実行されない" は表示されない
```

ステップ4.3: try-catch文の基本構造

try-catch文は、エラーが発生する可能性のあるコードを安全に実行するための制御構文です。tryブロック内でエラーが発生すると、プログラムを停止させる代わりに、catchブロックでエラー処理を行い、プログラムの実行を継続できます。

基本構文：

```
try {
    // エラーが発生する可能性のあるコード
} catch (error) {
    // エラーが発生した場合の処理
}
```

エラーハンドリングなしの場合：

```
console.log("処理開始");
let data = null;
console.log(data.length); // ここでエラーが発生してプログラム停止
console.log("この行は実行されない");
```

下のコードにtry-catch文を追加してプログラムが継続実行されるようにしてください：

```
console.log("処理開始");
___
    let data = null;
    console.log(data.length);
___ (error) {
    console.log("エラーを処理しました");
___
console.log("処理継続");
```

正解

```
console.log("処理開始");
try {
```

```
    let data = null;
    console.log(data.length);
} catch (error) {
    console.log("エラーを処理しました");
}
console.log("処理継続");
```

ステップ4.4: throw文による意図的なエラー発生

`throw`文は、プログラマーが意図的にエラーを発生させるための構文です。不正な条件を検出した場合に、適切なエラーメッセージと共にエラーを発生させ、呼び出し元に問題を通知することができます。

throw文の基本構文：

```
throw new Error("エラーメッセージ");
```

実際の使用例：

```
function divide(a, b) {
    if (b === 0) {
        throw new Error("0で割ることはできません");
    }
    return a / b;
}

try {
    let result = divide(10, 0); // エラーが発生
    console.log(result);
} catch (error) {
    console.log("計算エラー:", error.message);
}
```

ステップ4.5: 配列アクセスの安全性確保

配列の要素にアクセスする際の安全性を確保する方法を学習しましょう。

配列のインデックスアクセスでは、存在しない要素にアクセスしようとするとundefinedが返されます。undefinedのプロパティにアクセスしようとするとエラーが発生するため、事前にチェックが必要です。

安全でない例：

```
function getTaskTitle(index) {
    return tasks[index].title; // tasks[index]がundefinedの場合エラー
}
```

安全な例（if文による事前チェック）：

```
function getTaskTitle(index) {
    if (index >= 0 && index < tasks.length) {
        return tasks[index].title;
    } else {
        return "無効なインデックス";
    }
}
```

安全な例（try-catchによるエラー処理）：

```
function getTaskTitle(index) {
    try {
        if (index < 0 || index >= tasks.length) {
            throw new Error(`無効なインデックス: ${index}`);
        }
        return tasks[index].title;
    } catch (error) {
        console.log("エラー:", error.message);
        return null;
    }
}
```

ステップ4.6: タスク削除機能でのエラーハンドリング実装

実際のTodoアプリでエラーハンドリングを実装してみましょう。

要件：

- 無効なインデックスの場合はエラーを発生
- タスクが存在しない場合はエラーメッセージを表示
- 正常な削除の場合は成功メッセージを表示

```
function deleteTask(index) {
    ___ {
        if (index < 0 || index >= tasks.length) {
```

```
            ___ new Error("無効なインデックスです");
        }

        if (!tasks[index]) {
            ___ new Error("タスクが存在しません");
        }

        tasks.splice(index, 1);
        console.log("タスクを削除しました");

    } ___ (error) {
        console.log("削除エラー:", error.message);
    }
}
```

正解

```
function deleteTask(index) {
    try {
        if (index < 0 || index >= tasks.length) {
            throw new Error("無効なインデックスです");
        }

        if (!tasks[index]) {
            throw new Error("タスクが存在しません");
        }

        tasks.splice(index, 1);
        console.log("タスクを削除しました");

    } catch (error) {
        console.log("削除エラー:", error.message);
    }
}
```

ステップ4.7: エラーオブジェクトの理解

JavaScriptのエラーオブジェクトには、エラーに関する詳細な情報が含まれています。主要なプロパティとして `message` （エラーメッセージ）、 `name` （エラーの種類）、 `stack` （エラーが発生した場所の情報）があります。

エラーオブジェクトのプロパティ：

```
try {
    let result = nonExistentFunction();
} catch (error) {
    console.log("エラー名:", error.name);       // "ReferenceError"
    console.log("メッセージ:", error.message);  // "nonExistentFunction is not defined"
    console.log("スタック:", error.stack);      // 詳細な発生場所
}
```

カスタムエラーメッセージの活用：

```
function validateTaskInput(title) {
    try {
        if (!title) {
            throw new Error("タスクタイトルが入力されていません");
        }
        if (title.length > 100) {
            throw new Error("タスクタイトルが長すぎます（100文字以内）");
        }
        if (title.trim() === "") {
            throw new Error("空白のみのタイトルは無効です");
        }
        return true;
    } catch (error) {
        console.log("入力検証エラー:", error.___);
        return false;
    }
}
```

下の穴埋めを完成させてください：

正解

```
function validateTaskInput(title) {
    try {
        if (!title) {
            throw new Error("タスクタイトルが入力されていません");
        }
        if (title.length > 100) {
            throw new Error("タスクタイトルが長すぎます（100文字以内）");
        }
        if (title.trim() === "") {
            throw new Error("空白のみのタイトルは無効です");
        }
        return true;
    } catch (error) {
        console.log("入力検証エラー:", error.message);
```

```
        return false;
    }
}
```

ステップ4.8: エラーハンドリングのベストプラクティス

適切なエラーハンドリングにより、ユーザーにとって使いやすく、開発者にとってデバッグしやすいプログラムを作成できます。エラーメッセージは分かりやすく、具体的で、ユーザーが次に何をすべきかが明確になるように書きます。

良いエラーメッセージの例：

```
// 悪い例
throw new Error("エラー");

// 良い例
throw new Error("タスク番号は1から10の間で入力してください（入力値: 15）");
```

エラーハンドリングの原則：

1. **具体的なメッセージ**: 何が問題なのかを明確に
2. **解決方法の提示**: ユーザーが次に何をすべきかを示す
3. **ログ記録**: 開発者がデバッグできる情報を保存
4. **プログラム継続**: 可能な限りプログラムを継続実行

Section 5: Node.js CLIインターフェースの構築

ステップ5.1: Node.jsとモジュールシステムの理解

Node.jsは、JavaScriptをブラウザの外で実行できる環境です。様々な機能が「モジュール」として提供されており、`require`文を使って必要な機能を読み込みます。モジュールシステムにより、プログラムを機能ごとに分割し、再利用可能にできます。

Node.jsモジュールの種類：

1. **組み込みモジュール**: Node.jsに最初から含まれているモジュール

- `readline`: コマンドライン入力
- `fs`: ファイル操作
- `path`: パス操作

2. **サードパーティモジュール**: npmでインストールできる外部モジュール
   - `express`: Webサーバー
   - `lodash`: ユーティリティ関数

requireの基本構文：

```
const モジュール名 = require("モジュール名");
```

ステップ5.2: readlineモジュールの詳細理解

readlineは、Node.jsでコマンドライン入力を処理するための組み込みモジュールです。キーボードからの入力を受け取り、プログラムとユーザーが対話できるインターフェースを提供します。CLIアプリケーション開発において必須のモジュールです。

readlineの基本的な役割：

- キーボード入力の受け取り
- 入力完了時のコールバック実行
- 入出力ストリームの管理

requireによるモジュール読み込み：

```
const readline = require("readline");
```

下のreadlineインターフェース作成コードを完成させてください：

```
const readline = require("readline");

const rl = readline.___({
    input: process.___,     // キーボード入力
    output: process.___     // 画面出力
});
```

正解

```
const readline = require("readline");

const rl = readline.createInterface({
    input: process.stdin,    // キーボード入力
    output: process.stdout   // 画面出力
});
```

ステップ5.3: process.stdinとprocess.stdoutの理解

`process.stdin` と `process.stdout` は、Node.jsにおける標準入出力ストリームです。 `stdin` （Standard Input）はキーボードからの入力を、 `stdout` （Standard Output）は画面への出力を表します。これらは UNIX系OSの概念をJavaScriptで利用できるようにしたものです。

標準入出力の流れ：

```
ユーザー（キーボード入力）→ process.stdin → プログラム → process.stdout → 画面表示
```

実際の使用例：

```
// ユーザーが "Hello" と入力
// ↓ process.stdin で受け取り
// ↓ プログラムで処理
// ↓ process.stdout で出力
// → 画面に "こんにちは、Hello さん！" と表示
```

ステップ5.4: rl.questionメソッドの仕組み

`rl.question` は、ユーザーに質問を表示し、入力を待って、入力が完了したら指定した関数（コールバック関数）を実行するメソッドです。非同期処理の一種で、プログラムは入力を待っている間も他の処理を継続できます。

rl.questionの基本構文：

```
rl.question("質問メッセージ", (回答) => {
    // 回答を受け取った後の処理
});
```

動作の流れ：

1. 質問メッセージを画面に表示
2. ユーザーの入力を待機
3. Enterキーが押されたら入力完了
4. 入力内容をコールバック関数の引数として渡す
5. コールバック関数を実行

実際の例：

```
rl.question("お名前を入力してください: ", (name) => {
    console.log(`こんにちは、${name}さん！`);
});
```

ステップ5.5: mapメソッドの基本理解

`map`は、配列の各要素を変換して新しい配列を作成するメソッドです。元の配列は変更されず、各要素に対して指定した変換処理を行った結果として新しい配列が返されます。メニューの表示など、データの変換において頻繁に使用される重要なメソッドです。

mapの基本構文：

```
新しい配列 = 元の配列.map(要素 => 変換処理);
```

mapの基本例：

```
let numbers = [1, 2, 3];
let doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

インデックスも使用する例：

```
let fruits = ["りんご", "バナナ"];
let numberedFruits = fruits.map((fruit, index) => `${index + 1}. ${fruit}`);
console.log(numberedFruits); // ["1. りんご", "2. バナナ"]
```

下の穴埋めを完成させてください：

```
let items = ["タスク追加", "タスク削除"];
let menuItems = items.___((___, index) => `${___ + 1}. ${___}`);
```

正解

```
let items = ["タスク追加", "タスク削除"];
let menuItems = items.map((item, index) => `${index + 1}. ${item}`);
```

ステップ5.6: メニューシステムの設計

CLIアプリケーションでは、ユーザーが選択できる操作をメニュー形式で表示するのが一般的です。配列で選択肢を管理し、mapメソッドで番号付きに変換して表示することで、ユーザーにとって分かりやすいインターフェースを提供できます。

メニュー設計の原則：

1. **明確な選択肢**: 各項目の機能が一目で分かる
2. **論理的な順序**: 使用頻度や重要度に応じた配置
3. **終了オプション**: プログラムを安全に終了する手段

メニュー配列の作成：

```
const MENU_OPTIONS = [
    "タスク追加", "一覧(forEach)", "一覧(for)", "一覧(while)",
    "完了", "削除", "検索", "統計", "プロパティ", "終了"
];
```

番号付きメニューの表示：

```
// 配列を番号付きの文字列に変換
let menuText = MENU_OPTIONS.___((___, index) => `${index + 1}. ${___}`);
console.log(menuText.join("  "));
```

下の穴埋めを完成させてください：

正解

```
// 配列を番号付きの文字列に変換
let menuText = MENU_OPTIONS.map((item, index) => `${index + 1}. ${item}`);
console.log(menuText.join("  "));
```

ステップ5.7: joinメソッドの基本理解

`join`は、配列の要素を指定した区切り文字で結合して、一つの文字列を作成するメソッドです。メニューの表示や、データの出力において頻繁に使用される重要なメソッドです。

joinメソッドの基本構文：

```
配列.join("区切り文字")
```

joinメソッドの例：

```
let items = ["りんご", "バナナ", "オレンジ"];

console.log(items.join(", "));    // "りんご, バナナ, オレンジ"
console.log(items.join(" | "));   // "りんご | バナナ | オレンジ"
console.log(items.join("¥n"));    // りんご（改行）バナナ（改行）オレンジ
```

下の穴埋めを完成させてください：

```
let menuItems = ["1. タスク追加", "2. タスク削除"];
let menuString = menuItems.___("  ");
console.log(menuString); // "1. タスク追加  2. タスク削除" と表示される
```

正解

```
let menuItems = ["1. タスク追加", "2. タスク削除"];
let menuString = menuItems.join("  ");
console.log(menuString); // "1. タスク追加  2. タスク削除" と表示される
```

ステップ5.8: mapとjoinを組み合わせたメニュー表示

mapメソッドとjoinメソッドを組み合わせることで、配列から見やすいメニュー
を作成できます。

```
function showMenu() {
    // 配列を番号付きの文字列に変換してから結合
    let menuString = MENU_OPTIONS.___((item, i) => `${i + 1}. ${item}`).___("  ");
    console.log("¥n" + menuString);

    rl.question("操作を選んでください: ", (answer) => {
        handleMenuSelection(answer.trim());
    });
}
```

下の穴埋めを完成させてください：

正解

```
function showMenu() {
    // 配列を番号付きの文字列に変換してから結合
    let menuString = MENU_OPTIONS.map((item, i) => `${i + 1}. ${item}`).join("  ");
    console.log("¥n" + menuString);

    rl.question("操作を選んでください: ", (answer) => {
        handleMenuSelection(answer.trim());
    });
}
```

ステップ5.9: trimメソッドの重要性

`trim`は、文字列の前後にある空白文字（スペース、タブ、改行など）を除去するメソッドです。ユーザー入力の処理において、意図しない空白による問題を防ぐために重要です。

trimが必要な理由：

```
// ユーザーが間違えて ” 1 ” と入力（前後にスペース）
let userInput = ” 1 ”;

console.log(userInput === ”1”);        // false（空白があるため）
console.log(userInput.trim() === ”1”); // true（空白が除去される）
```

下の穴埋めを完成させてください：

```
rl.question(”番号を選択: ”, (answer) => {
    let cleanAnswer = answer.___(); // 前後の空白を除去
    handleMenuSelection(cleanAnswer);
});
```

正解

```
rl.question(”番号を選択: ”, (answer) => {
    let cleanAnswer = answer.trim(); // 前後の空白を除去
    handleMenuSelection(cleanAnswer);
});
```

ステップ5.10: switch文による多分岐処理

switch文は、一つの変数の値に対して複数の条件分岐を効率的に処理するための制御構文です。多数のif-else文を書くよりも読みやすく、メニュー選択処理のような場面で威力を発揮します。各case文の最後にはbreak文が必要です。

switch文の基本構造：

```
switch (変数) {
    case ”値1”:
```

```
        // 処理1
        break;
    case ”値2”:
        // 処理2
        break;
    default:
        // どの値にも一致しない場合
}
```

break文の重要性：

```
switch (value) {
    case ”1”:
        console.log(”1が選択”);
        // breakがないと次のcaseも実行される
    case ”2”:
        console.log(”2が選択”);
        break;
}
```

下の多分岐をswitch文で実装してください：

```
function processNumber(num) {
    ___ (num) {
        ___ ”1”:
            console.log(”一”);
            ___;
        ___ ”2”:
            console.log(”二”);
            ___;
        ___:
            console.log(”その他”);
    }
}
```

正解

```
function processNumber(num) {
    switch (num) {
        case ”1”:
            console.log(”一”);
            break;
        case ”2”:
            console.log(”二”);
```

```
            break;
        default:
            console.log(”その他”);
    }
}
```

ステップ5.11：非同期処理とコールバック関数

JavaScriptの非同期処理では、処理の完了を待たずに次の処理に進みます。
`rl.question` は非同期関数で、ユーザーの入力完了後にコールバック関数が実行されます。連続した入力が必要な場合は、コールバック関数の中で次の `rl.question` を呼び出す入れ子構造になります。

コールバック関数の仕組み：

```
console.log(”1. プログラム開始”);

rl.question(”質問1: ”, (answer1) => {
    console.log(”3. 回答1を受け取り:”, answer1);

    rl.question(”質問2: ”, (answer2) => {
        console.log(”5. 回答2を受け取り:”, answer2);
    });

    console.log(”4. 質問2を表示”);
});

console.log(”2. 質問1を表示”);
```

実行順序：

1. プログラム開始
2. 質問1を表示
3. 回答1を受け取り
4. 質問2を表示
5. 回答2を受け取り

ステップ5.12：連続入力処理の実装

複数の項目を順番に入力する処理を実装しましょう。

```
function promptAddTask() {
    rl.question("タスク名: ", (title) => {
        rl.question("説明: ", (desc) => {
            rl.question("優先度(数字): ", (priority) => {
                rl.question("重要? (true/false): ", (important) => {
                    // すべての入力が完了した時点で実行
                    const newTask = createTask(
                        title,
                        desc,
                        priority,
                        important === "___"
                    );
                    tasks.push(newTask);
                    console.log(`タスクを追加しました: ${newTask.title}`);
                    showMenu();
                });
            });
        });
    });
}
```

下の穴埋めを完成させてください：

正解

```
function promptAddTask() {
    rl.question("タスク名: ", (title) => {
        rl.question("説明: ", (desc) => {
            rl.question("優先度(数字): ", (priority) => {
                rl.question("重要? (true/false): ", (important) => {
                    // すべての入力が完了した時点で実行
                    const newTask = createTask(
                        title,
                        desc,
                        priority,
                        important === "true"
                    );
                    tasks.push(newTask);
                    console.log(`タスクを追加しました: ${newTask.title}`);
                    showMenu();
                });
            });
        });
    });
}
```

ステップ5.13: rl.close()によるプログラム終了

`rl.close()`は、readlineインターフェースを閉じ、プログラムを正常に終了させるメソッドです。CLIアプリケーションでは、ユーザーが「終了」を選択した時に呼び出し、リソースを適切に解放します。

プログラム終了の流れ：

1. ユーザーが「10」（終了）を選択
2. `rl.close()`を実行
3. readlineインターフェースが閉じられる
4. Node.jsプロセスが終了
5. コマンドプロンプトに戻る

適切な終了処理：

```
case "10":
    console.log("アプリケーションを終了します...");
    rl.___();
    break;
```

下の穴埋めを完成させてください：

正解

```
case "10":
    console.log("アプリケーションを終了します...");
    rl.close();
    break;
```

Section 6: オブジェクト操作とプロパティアクセスの詳細

ステップ6.1: typeof演算子の基本理解

typeof演算子は、変数や値のデータ型を文字列で返す演算子です。プログラム実行時に型を確認し、型に応じた適切な処理を行うために使用します。エラーを防

ぎ、安全なプログラムを作成するための重要なツールです。

typeof演算子の基本構文：

```
typeof 変数名
typeof 値
```

主要な型の確認：

```
console.log(typeof 25);        // "number"
console.log(typeof "text");    // "string"
console.log(typeof true);      // "boolean"
console.log(typeof undefined); // "undefined"
```

下の型確認コードを完成させてください：

```
let age = 30;
let name = "田中";
let isActive = false;

console.log("age の型:", ___ age);
console.log("name の型:", ___ name);
console.log("isActive の型:", ___ isActive);
```

正解

```
let age = 30;
let name = "田中";
let isActive = false;

console.log("age の型:", typeof age);
console.log("name の型:", typeof name);
console.log("isActive の型:", typeof isActive);
```

ステップ6.2: 入力値の型チェックと変換

ユーザーからの入力は常に文字列として受け取られます。数値として使用する場合は、事前に型変換を行い、変換が正常に行われたかをチェックする必要があり

ます。

安全な数値変換の例：

```
function safeNumberConversion(input) {
    // 文字列を数値に変換
    let number = Number(input);

    // NaN（変換失敗）をチェック
    if (typeof number === "number" && !isNaN(number)) {
        return number;
    } else {
        return null; // 変換失敗
    }
}

// 使用例
let userInput = "25";
let taskIndex = safeNumberConversion(userInput); // 25
let invalidInput = safeNumberConversion("abc"); // null
```

タスク番号の安全な処理：

```
function processTaskIndex(input) {
    let index = Number(input) - 1; // 1ベースから0ベースに変換

    if (___ index !== "number" || isNaN(index)) {
        return null; // 数値でない
    }

    if (index < 0 || index >= tasks.length) {
        return null; // 範囲外
    }

    return index; // 有効なインデックス
}
```

下の穴埋めを完成させてください：

正解

```
function processTaskIndex(input) {
    let index = Number(input) - 1; // 1ベースから0ベースに変換
```

```
    if (typeof index !== "number" || isNaN(index)) {
        return null; // 数値でない
    }

    if (index < 0 || index >= tasks.length) {
        return null; // 範囲外
    }

    return index; // 有効なインデックス
}
```

ステップ6.3: Object.entries()の理解と活用

Object.entries()は、オブジェクトの各プロパティを［キー, 値］の配列形式で返すメソッドです。オブジェクトの全プロパティを順番に処理したい場合に使用します。返される配列は、プロパティの数だけ要素を持ちます。

基本的な動作：

```
let task = {
    title: "買い物",
    completed: false
};

let entries = Object.entries(task);
console.log(entries);
// [ ["title", "買い物"], ["completed", false] ]
```

配列の各要素は［キー, 値］の形式：

```
console.log(entries[0]); // ["title", "買い物"]
console.log(entries[1]); // ["completed", false]
```

下のオブジェクトをObject.entries()で変換してください：

```
let user = {
    id: 1,
    name: "山田"
};

let userEntries = Object.___(user);
console.log(userEntries); // [ ["id", 1], ["name", "山田"] ] と表示される
```

正解

```
let user = {
    id: 1,
    name: "山田"
};

let userEntries = Object.entries(user);
console.log(userEntries); // [ ["id", 1], ["name", "山田"] ] と表示される
```

ステップ6.4: 分割代入（Destructuring）の理解

**分割代入は、配列から要素を取り出して複数の変数に一度に代入する構文です。
Object.entries()が返す［キー, 値］の配列に対して使用することで、キーと値
を別々の変数として同時に取得できます。**

配列の分割代入の基本：

```
let data = ["apple", "red"];
let [name, color] = data;
console.log(name);  // "apple"
console.log(color); // "red"
```

従来の方法と分割代入の比較：

```
let entry = ["title", "買い物"];

// 従来の方法
let key = entry[0];
let value = entry[1];

// 分割代入
let [key, value] = entry;
```

下の配列を分割代入で取り出してください：

```
let userInfo = ["田中", 30];
let [___, ___] = userInfo;
```

```
console.log(`名前: ${userName}, 年齢: ${userAge}`);
```

正解

```
let userInfo = ["田中", 30];
let [userName, userAge] = userInfo;
console.log(`名前: ${userName}, 年齢: ${userAge}`);
```

ステップ6.5: 学習内容の統合演習

Object.entries()、分割代入、typeof演算子を組み合わせて、プロパティ表示機能を実装します。

複数の概念を組み合わせることで、より実用的で高機能なプログラムを作成できます。Object.entries()でプロパティを取得し、分割代入でキーと値を分離し、typeof演算子で型情報を表示する総合的な機能を実装します。

統合する3つの要素：

1. Object.entries() - オブジェクトのプロパティを配列で取得
2. 分割代入 - [キー, 値] を別々の変数に代入
3. typeof演算子 - 値の型を確認

下の関数を完成させてください：

```
function showTaskProperties(index) {
    if (typeof index !== "number" || index < 0 || index >= tasks.length) {
        console.log("無効なインデックスです");
        return;
    }

    const task = tasks[index];
    console.log("タスクのプロパティ:");

    // 3つの要素を全て使って実装
    for (const [___, ___] of Object.___(___)) {
        console.log(`  ${key}: ${value} (型: ${___ value})`);
    }
}
```

正解

```
function showTaskProperties(index) {
    if (typeof index !== "number" || index < 0 || index >= tasks.length) {
        console.log("無効なインデックスです");
        return;
    }

    const task = tasks[index];
    console.log("タスクのプロパティ:");

    // 3つの要素を全て使って実装
    for (const [key, value] of Object.entries(task)) {
        console.log(`  ${key}: ${value} (型: ${typeof value})`);
    }
}
```

ステップ6.6: プロパティ表示の入力処理実装

ユーザーからの入力を受け取って、プロパティ表示機能を呼び出す処理を実装しましょう。

```
function promptShowProperties() {
    rl.question("プロパティを表示するタスク番号: ", (num) => {
        const index = Number(num) - 1;
        showTaskProperties(index);
        showMenu();
    });
}
```

Section 7: 完全なアプリケーションの統合

ステップ7.1: タスク完了機能の実装

タスク完了機能では、指定されたタスクの `completed` プロパティを `true` に変更します。既に完了済みのタスクや、存在しないタスクに対する適切なエラーハンドリングも重要です。

タスク完了機能の要件：

1. 指定されたインデックスのタスクが存在するかチェック
2. すでに完了済みでないかチェック
3. 条件を満たす場合のみ完了状態に変更
4. 適切なフィードバックメッセージを表示

```
function completeTask(index) {
    // タスクの存在確認と未完了確認
    if (tasks[index] && !tasks[index].completed) {
        tasks[index].___ = true;
        console.log("タスクを完了しました！");
    } else if (tasks[index] && tasks[index].completed) {
        console.log("このタスクは既に完了済みです");
    } else {
        console.log("無効なタスク番号です");
    }
}
```

下の穴埋めを完成させてください：

正解

```
function completeTask(index) {
    // タスクの存在確認と未完了確認
    if (tasks[index] && !tasks[index].completed) {
        tasks[index].completed = true;
        console.log("タスクを完了しました！");
    } else if (tasks[index] && tasks[index].completed) {
        console.log("このタスクは既に完了済みです");
    } else {
        console.log("無効なタスク番号です");
    }
}
```

ステップ7.2: 各種入力処理関数の実装

各機能に対応する入力処理関数を実装します。

```
// タスク完了の入力処理
function promptCompleteTask() {
    rl.question("完了にするタスク番号: ", (num) => {
```

```
        const index = Number(num) - 1;
        completeTask(index);
        showMenu();
    });
}

// タスク削除の入力処理
function promptDeleteTask() {
    rl.question("削除するタスク番号: ", (num) => {
        const index = Number(num) - 1;
        deleteTask(index);
        showMenu();
    });
}

// タスク検索の入力処理
function promptSearchTask() {
    rl.question("検索ワード: ", (word) => {
        searchTasks(word);
        showMenu();
    });
}
```

ステップ7.3: メニュー処理の完全版

すべての機能をメニューに統合しましょう。

```
function handleMenuSelection(selection) {
    switch (selection) {
        case "1":
            promptAddTask();
            break;
        case "2":
            listTasksForEach();
            showMenu();
            break;
        case "3":
            listTasksFor();
            showMenu();
            break;
        case "4":
            listTasksWhile();
            showMenu();
            break;
        case "5":
            promptCompleteTask();
            break;
        case "6":
            promptDeleteTask();
            break;
        case "7":
            promptSearchTask();
```

```
            break;
        case "8":
            showStatistics();
            showMenu();
            break;
        case "9":
            promptShowProperties();
            break;
        case "10":
            console.log("アプリケーションを終了します...");
            rl.close();
            break;
        default:
            console.log("1〜10の番号で選択してください");
            showMenu();
    }
}
```

ステップ7.4: アプリケーションの初期化と起動

アプリケーションのエントリーポイントは、プログラムが開始される最初の処理部分です。グローバル変数の初期化、ウェルカムメッセージの表示、メイン処理の開始を順番に行います。ユーザーが最初に見る部分なので、分かりやすい説明が重要です。

初期化処理の順序：

1. 必要な変数の宣言
2. ウェルカムメッセージの表示
3. メイン処理の開始

下のアプリケーション起動コードを完成させてください：

```
// 1. グローバル変数の初期化
let tasks = [];

// 2. ウェルカムメッセージ
console.log("=== Todo CLI アプリケーション ===");
console.log("JavaScript を使った実践的な CLI ツール");

// 3. メイン処理開始
___();
```

正解

```
// 1. グローバル変数の初期化
let tasks = [];

// 2. ウェルカムメッセージ
console.log("=== Todo CLI アプリケーション ===");
console.log("JavaScript を使った実践的な CLI ツール");

// 3. メイン処理開始
showMenu();
```

ステップ7.5: 完全なファイル構成の確認

完成したアプリケーションのファイル構成を確認しましょう。

ファイルの構成順序：

1. **モジュールのインポート**: `require`文
2. **定数の定義**: `MENU_OPTIONS`など
3. **グローバル変数**: `tasks`配列
4. **ユーティリティ関数**: `createTask`など
5. **メイン機能関数**: `listTasks`, `deleteTask`など
6. **入力処理関数**: `promptAddTask`など
7. **メニュー関数**: `showMenu`, `handleMenuSelection`
8. **アプリケーション起動**: メイン処理の開始

この構成により、読みやすく保守しやすいコードが完成します。

✅ **完成確認**

これですべての機能が実装されました。以下のコマンドでアプリケーションを実行できます：

```
node todo.js
```

まとめと学習成果

習得したJavaScript技術要素

1. **基本構文**
   - 変数宣言（let, const）
   - データ型（number, string, boolean, object, array）
   - 関数定義と呼び出し
2. **配列操作**
   - forEach, map, filter, every, some
   - push, splice, length
3. **制御構文**
   - if/else条件分岐
   - for, while, forEach繰り返し
   - switch文による多分岐
4. **オブジェクト指向**
   - オブジェクトリテラル
   - プロパティアクセス
   - Object.entries()
5. **エラーハンドリング**
   - try-catch文
   - Error オブジェクト
6. **Node.js特有機能**
   - require文によるモジュール読み込み
   - readline による対話的入力
   - process.stdin/stdout

次のステップへ

- **上級JavaScript**: async/await, Promise, ES6+機能
- **Webアプリケーション**: DOM操作, イベント処理
- **フレームワーク学習**: React, Vue.js, Express.js
- **データベース連携**: MongoDB, MySQL との接続

完成版コード

▶ todo.js 完全版（クリックして表示）
　これで、JavaScript の基礎から実践的なCLIアプリケーション開発まで、体系的に学習することができました！

Section 8: デバッグとトラブルシューティング

ステップ8.1: console.logの効果的な使い方

`console.log` は、プログラムの動作を確認するための最も基本的なデバッグツールです。変数の値、関数の実行タイミング、エラーの原因を特定するために使用します。効果的に使うことで、問題を素早く発見し、解決できます。

デバッグのための `console.log` テクニック:

```
// 変数の値を確認
let userName = "田中";
console.log("ユーザー名:", userName);

// 関数の開始と終了を確認
function addTask(title) {
    console.log("addTask関数開始 - タイトル:", title);
    // 処理...
    console.log("addTask関数終了");
}

// オブジェクトの中身を確認
let task = {title: "買い物", completed: false};
console.log("タスクオブジェクト:", task);
```

ステップ8.2: よくあるエラーとその解決法

プログラミングを始めたばかりの時は、よく似たエラーに遭遇します。これらのエラーパターンを覚えておくことで、同じ問題に直面した時に迅速に解決できます。

頻出エラー一覧:

エラーメッセージ	原因	解決方法
`ReferenceError: xxx is not defined`	変数や関数が宣言されていない	変数の宣言を確認、スペルミスをチェック
`TypeError: Cannot read property 'xxx' of undefined`	オブジェクトがundefinedまたはnull	オブジェクトの存在確認を追加
`SyntaxError: Unexpected token`	文法エラー（括弧の不一致など）	括弧、クォートの対応を確認
`TypeError: xxx.push is not a function`	配列でない変数に配列メソッドを使用	変数の型を確認

実際のトラブルシューティング例:

```
// エラーが出る例
function displayTask(index) {
    console.log(tasks[index].title); // エラー：tasks[index]がundefined
}

// 修正版
function displayTask(index) {
    if (index >= 0 && index < tasks.length && tasks[index]) {
        console.log(tasks[index].title);
    } else {
        console.log("無効なタスク番号です");
    }
}
```

Section 9: コードの改善と保守

ステップ9.1: 読みやすいコードの書き方

プログラムは書いた本人だけでなく、他の人も読むことを前提に書きます。また、数か月後の自分も「他の人」です。読みやすいコードを書くことで、バグの発見、機能追加、チーム開発が格段に楽になります。

読みやすいコードのルール:

1. **意味のある変数名**

```
// 悪い例
let d = new Date();
let u = "田中";

// 良い例
let currentDate = new Date();
let userName = "田中";
```

2. **適切なコメント**

```
// タスクの完了状態を切り替える関数
function toggleTaskCompletion(taskIndex) {
    // 配列の範囲内かチェック
    if (taskIndex >= 0 && taskIndex < tasks.length) {
        tasks[taskIndex].completed = !tasks[taskIndex].completed;
    }
}
```

3. **関数の分割**

```
// 長すぎる関数は分割する
function processUserInput(input) {
    let cleanInput = sanitizeInput(input);
    let validInput = validateInput(cleanInput);
    return processValidInput(validInput);
}
```

ステップ9.2: 機能追加の考え方

既存のプログラムに新しい機能を追加する時は、既存のコードの構造を理解し、一貫性を保ちながら実装することが重要です。小さな変更から始めて、徐々に複雑な機能を追加していきます。

機能追加の手順:

1. **要件の明確化**: 何を実現したいかを具体的に定義
2. **既存コードの分析**: 似た機能がないか確認
3. **設計**: どこにどんなコードを追加するか計画
4. **実装**: 小さな部分から実装
5. **テスト**: 新機能と既存機能の動作確認

実例：タスクの優先度変更機能を追加

```
// 既存の構造に合わせて新機能を追加
function changePriority(taskIndex, newPriority) {
    try {
        if (taskIndex >= 0 && taskIndex < tasks.length) {
            tasks[taskIndex].priority = Number(newPriority);
            console.log("優先度を変更しました");
        } else {
            throw new Error("無効なタスク番号です");
        }
    } catch (error) {
        console.log("エラー:", error.message);
    }
}
```

Section 10: 次のステップへの道筋

ステップ10.1: 学習の継続方法

プログラミングの学習は継続が重要です。基礎を固めた後は、実際のプロジェクト作成、新しい技術の習得、コミュニティへの参加を通じて、継続的にスキルアップしていきます。

推奨学習ステップ:

1. **基礎の反復練習**
   - 今回のTodoアプリに機能追加
   - 別のCLIアプリケーション作成
   - アルゴリズムの問題解決
2. **Web開発への展開**
   - HTML/CSSの学習
   - DOM操作とイベント処理
   - Webページでの動的な処理
3. **フレームワークの学習**
   - React.js（ユーザーインターフェース）

- Node.js + Express（サーバーサイド）
- データベース連携

4. **実践的なプロジェクト**
   - ポートフォリオサイト作成
   - 小規模なWebアプリケーション開発
   - オープンソースプロジェクトへの貢献

ステップ10.2: 実務で活用できるスキル

今回学習した内容は、実際の業務で以下のような形で活用できます。プログラミングの知識は、エンジニア以外の職種でも、作業の自動化や効率化に役立ちます。

業務での活用例:

1. **作業自動化**
   - Excelデータの処理自動化
   - ファイル整理のスクリプト作成
   - 定期的なレポート生成
2. **データ分析**
   - ログファイルの解析
   - 売上データの集計
   - 顧客情報の整理
3. **ツール開発**
   - 社内用の管理ツール
   - 簡単な計算ツール
   - データ変換ツール

継続学習のためのリソース:

- **公式ドキュメント**: MDN Web Docs (JavaScript)
- **練習サイト**: LeetCode, Codewars
- **コミュニティ**: Qiita, Stack Overflow
- **書籍**: JavaScript関連の技術書

ステップ10.3: プログラミング習慣の確立

プログラミングスキルの向上には、継続的な練習が不可欠です。毎日少しずつでも、コードを書く習慣を身につけることで、着実にスキルアップできます。

効果的な学習習慣:

1. **毎日のコーディング**
   - 1日30分からでも毎日コードを書く
   - 小さな課題から始める
   - 学習記録を付ける
2. **コードレビューの習慣**
   - 自分のコードを見直す
   - 他の人のコードを読む
   - 改善点を見つける練習
3. **実践的なプロジェクト**
   - 学習した内容を組み合わせたアプリ作成
   - 実際の問題を解決するツール開発
   - ポートフォリオの充実

🎉 学習完了おめでとうございます！

真のプログラマーへの道のりは今から始まります。今回のガイドで学習した基礎知識を土台に、継続的な学習と実践を通じて、さらなるスキルアップを目指してください！

最終課題: 独自機能の追加 今回作成したTodoアプリに、以下のような独自機能を追加してみましょう：

- タスクの期限設定機能
- カテゴリ別フィルター
- タスクのエクスポート機能
- ユーザー設定の保存

プログラミングの世界はとても広く、常に新しい技術が生まれています。基礎をしっかりと身につけた今、どんな技術にも挑戦できる準備が整いました！