ステップ5: タスク削除時のlocalStorageからの削除（CRUD操作完成）（穴埋め版）

導入・問題提起

前回のステップ4で「タスクの完了状態保存」機能を実装しました。 現在、タスクの追加・完了状態変更はlocalStorageに保存されますが、既にdeleteTask()関数にも保存機能が含まれています。

今回は、deleteTask()関数の動作を詳しく確認し、削除操作も含めた **完全なCRUD操作の永続化** について学習します。

現在の状況確認:

- タスクを追加 → ✅ localStorage保存される
- タスクの完了切り替え → ✅ localStorage保存される（ステップ4で実装）
- タスクを削除 → ✅ **実は既にlocalStorage保存されている**
- ページ再読み込み → ✅ 削除も含めて正確に復元される

このステップの目標

- タスク削除操作もlocalStorageに正しく反映されることを理解する
- データの整合性を保つ重要性を学ぶ
- CRUD操作（作成・読み取り・更新・削除）の全てでlocalStorage同期が必要であることを理解する
- Webアプリケーションにおける完全なデータ永続化を習得する

考えてみよう

手順1: deleteTask関数の動作確認

現在のdeleteTask関数を確認し、localStorage保存機能が含まれていることを学習します。

```
function deleteTask(id) {
    // 削除前にタスク情報を取得（ログ出力用）
    const taskToDelete = tasks.find(task => task.id === id);

    // 指定されたIDのタスクを配列から削除
    tasks = tasks.filter(function (task) {
        return task.id !== id;
    });

    console.log(`🗑️ タスク「${taskToDelete.text}」を削除しました`);

    // 【確認①】削除をlocalStorageにも反映する関数呼び出し（20文字）
    // 答え: ____________________
    ____________________;

    // 画面表示を更新
    renderTasks();
    updateProgress();

    console.log("💾 削除操作をlocalStorageに保存しました");
}
```

手順2: CRUD操作の完全性確認

Webアプリケーションの基本操作（CRUD）全てでlocalStorage同期が実現されていることを確認します。

```
// 【確認②】以下のCRUD操作とlocalStorage保存の対応を確認してください

// Create（作成）: addTask() → ?
// Read（読み取り）: ? → JSON.parse()
// Update（更新）: toggleTask() → ?
// Delete（削除）: deleteTask() → ?

// 答え:
// Create（作成）: addTask() → ____________________（20文字）
// Read（読み取り）: ____________________（25文字） → JSON.parse()
// Update（更新）: toggleTask() → ____________________（20文字）
// Delete（削除）: deleteTask() → ____________________（20文字）
```

手順3: 動作確認テスト

実装を理解した後、以下の手順で完全な永続化を確認します。

1. **複数タスクを追加**（例：「買い物」「掃除」「勉強」）
2. **一部を完了状態**にする（例：「掃除」をチェック）
3. **一つを削除**する（例：「買い物」を削除）

4. **DevToolsのConsole**で削除メッセージを確認
5. **ページを再読み込み**
6. **結果確認**：削除したタスクが復活せず、完了状態も維持される

✨ 最小限ヒント

- このステップでは、既存のコードに大きな変更を加える必要はありません。主に確認と理解が中心です。
- `deleteTask`関数内で`saveTasksToStorage()`が呼び出されていることを確認します。これにより、タスク配列から特定のタスクが削除された後、その変更がlocalStorageに保存されます。
- CRUD操作とは、Create（作成）、Read（読み取り）、Update（更新）、Delete（削除）の頭文字を取ったもので、データ操作の基本的な機能群を指します。
  - **C**reate: `addTask()`関数が新しいタスクを作成し、`saveTasksToStorage()`で保存します。
  - **R**ead: `loadTasksFromStorage()`関数がページ読み込み時にlocalStorageからデータを読み取ります。
  - **U**pdate: `toggleTask()`関数がタスクの完了状態を更新し、`saveTasksToStorage()`で保存します。
  - **D**elete: `deleteTask()`関数がタスクを削除し、`saveTasksToStorage()`で保存します。

解答例（確認ポイント）

手順1: deleteTask関数の確認

```
function deleteTask(id) {
    // ...（既存のコード）...

    // 【確認①】削除をlocalStorageにも反映する関数呼び出し
    saveTasksToStorage(); // この呼び出しがあることを確認

    // ...（既存のコード）...
}
```

手順2: CRUD操作の完全性確認

```
// Create（作成）: addTask() → saveTasksToStorage()
// Read（読み取り）: loadTasksFromStorage() → JSON.parse()（関数内部で実行）
// Update（更新）: toggleTask() → saveTasksToStorage()
// Delete（削除）: deleteTask() → saveTasksToStorage()
```

ポイント解説

① CRUD操作とlocalStorageの完全同期

この演習シリーズを通じて、タスク管理アプリケーションにおける基本的なデータ操作であるCRUD（Create, Read, Update, Delete）のそれぞれが、localStorageと同期されるようになりました。

- **Create**: `addTask`関数で新しいタスクが作られると、`saveTasksToStorage`によってlocalStorageに保存されます。
- **Read**: アプリケーション（ページ）読み込み時に`loadTasksFromStorage`がlocalStorageからデータを読み込み、タスクリストを復元します。
- **Update**: `toggleTask`関数でタスクの完了状態が変更されると、`saveTasksToStorage`によってその変更がlocalStorageに保存されます。
- **Delete**: `deleteTask`関数でタスクが削除されると、`saveTasksToStorage`によってその削除がlocalStorageに反映されます。 これにより、ユーザーが行った全ての変更がブラウザを閉じても失われず、次回起動時にも保持される、一貫性のある永続化が実現できています。

② データ整合性の重要性

アプリケーションが表示している情報（画面上のタスクリスト）と、永続化されているデータ（localStorage内のタスクデータ）が常に一致している状態を「データ整合性が保たれている」と言います。`saveTasksToStorage`を各操作の適切なタイミングで呼び出すことにより、この整合性を維持しています。

③ Webアプリケーションにおける状態管理の基礎

この一連のステップは、Webアプリケーションにおけるクライアントサイドの状態管理の基本的なアプローチを示しています。ユーザー操作によって変化するデータを適切にメモリ上で管理し、それをブラウザのストレージ機能と同期させることで、よりリッチで実用的なユーザー体験を提供できます。

補足資料への誘導

- **localStorage APIの全体像**: `補足資料/localStorage_API詳解.md`
- **複雑なデータ構造の扱い**: `補足資料/データ構造設計.md`
- **他のブラウザストレージ技術との比較**: `補足資料/ブラウザストレージ比較.md`

動作確認チェックリスト

全てのステップが完了した状態で、以下の総合的な動作確認を行いましょう。

1. **タスクを3つ追加**します（例: Task A, Task B, Task C）。
   - ☐ 各タスクがリストに表示される。
   - ☐ localStorageに3つのタスクが保存されている。
2. **Task B のチェックボックスをクリック**して完了状態にします。
   - ☐ Task B の表示が完了状態に変わる。
   - ☐ localStorage内のTask Bの `completed` が `true` になっている。
   - ☐ 進捗バーが更新される。
3. **Task A の削除ボタンをクリック**して削除します。
   - ☐ Task A がリストから消える。
   - ☐ localStorageからTask Aが削除され、Task BとTask Cのみが残っている。
   - ☐ 進捗バーが更新される。
4. **ページを再読み込み（F5キー）**します。
   - ☐ Task A は表示されない（削除されたままである）。
   - ☐ Task B は完了状態で表示される。
   - ☐ Task C は未完了状態で表示される。
   - ☐ localStorageの内容は再読み込み前と変わらない（Task Bが完了、Task Cが未完了）。
   - ☐ 進捗バーが現在の状態（Task B完了、Task C未完了）を正しく反映している。
5. **残っているタスク（Task B, Task C）を全て削除**します。
   - ☐ リストが空になる。
   - ☐ localStorageからも全てのタスクが削除される。
   - ☐ 進捗バーが0%になる。
6. **再度ページを再読み込み**します。
   - ☐ リストは空のまま表示される。
   - ☐ localStorageも空のままである。

これらの確認を通じて、CRUD操作全てにおいてデータが正しく永続化され、
アプリケーションが期待通りに動作することを検証します。

全体振り返りと次のステップ

この演習シリーズで学んだこと

1. **ステップ1**: localStorageの基本的なAPI操作（`setItem`, `getItem`, `removeItem`, `clear`）と、オブジェクトを保存するための`JSON.stringify`/`JSON.parse`の必要性を学びました。
2. **ステップ2**: タスクが追加された際に、`tasks`配列をJSON文字列に変換してlocalStorageに自動保存する`saveTasksToStorage`関数を実装し、`addTask`関数と連携させました。また、`try...catch`によるエラーハンドリングの基本を導入しました。
3. **ステップ3**: ページ読み込み時にlocalStorageからタスクデータを読み込み、JSONをパースして`tasks`配列を復元し、画面に再表示する`loadTasksFromStorage`関数を実装しました。`DOMContentLoaded`イベントリスナー内でこれを呼び出すことで、初期表示を実現しました。
4. **ステップ4**: タスクの完了状態（`completed`プロパティ）が変更された際に、`toggleTask`関数内で`saveTasksToStorage`を呼び出し、変更を永続化しました。
5. **ステップ5**: `deleteTask`関数でも同様に`saveTasksToStorage`が呼び出されることを確認し、タスクの削除操作もlocalStorageに反映されることを学びました。これにより、CRUD操作全てにおけるデータ永続化が完成しました。

おめでとうございます！これで、localStorageを利用してToDoリストのデータを永続化する基本的な機能を全て実装できました。

今後の学習のために

- **より高度なエラーハンドリング**: localStorageの容量制限超過など、さまざまなエラーケースへの対応。
- **パフォーマンス**: 大量データを扱う場合のlocalStorageのパフォーマンス特性と代替手段（IndexedDBなど）。
- **UI/UXの改善**: より洗練されたユーザーインターフェースやユーザー体験の追求。
- **外部API連携**: サーバーサイドと連携してデータを同期する方法。
- **フレームワーク/ライブラリの活用**: React, Vue, Angularなどの現代的なJavaScriptフレームワーク/ライブラリを使った状態管理。