Tutorial Script 3: Using Claude to Understand a New Codebase

Duration: 10-12 minutes **Objective:** Demonstrate a systematic approach to learning a new codebase using Claude Code, from high-level architecture to specific implementation details

---

INTRODUCTION (1 minute)

[SCREEN: Terminal showing a fresh directory]

Professor:

"Welcome back! In the previous tutorials, we learned how to set up Claude Code and how to use it effectively. Now we're going to put those skills to work in a real-world scenario.

Imagine you've just joined a development team, or you're contributing to an open-source project. You're faced with thousands of lines of code across dozens of files. Where do you even start?

This is where Claude Code becomes invaluable. Today, I'll show you a systematic approach to understanding a new codebase using Claude as your guide.

We'll use our minimal-flashcards application, but this approach works for ANY codebase. Let's get started!"

PART 1: Clone the Repository and Initialize (2 minutes)

[SCREEN: Terminal]

Professor:

"Step one: Get the code and set up Claude. Let me start from scratch so you can see the complete workflow.

First, clone the repository:"

```
cd ~/Documents/learning
git clone <repository-url> minimal-flashcards
cd minimal-flashcards
```

Professor:

"Now, before we start asking Claude questions, we should initialize it. This is crucial - remember the /init command from our last tutorial?"

```
claude
```

[SCREEN: Claude Code interface opens]

[TYPE in Claude Code]:

```
/init
```

[SCREEN: Shows Claude analyzing the codebase]

Professor:

"Watch what happens. Claude is:

1. Exploring the directory structure
2. Reading key files like README.md and package.json
3. Identifying the technology stack
4. Understanding the project architecture
5. Creating the CLAUDE.md file

This takes about 30 seconds to a minute, but it's building a comprehensive understanding of the project.

Let's see what it created:"

[SCREEN: Open CLAUDE.md in editor]

Professor:

"Perfect! Claude has documented:

- The tech stack (React, FastAPI, PostgreSQL)
- Common development commands
- Architecture patterns
- Testing approaches

This is now Claude's 'reference guide' for our codebase. Every question we ask, Claude will have this context in mind.

Now we're ready to start exploring!"

PART 2: Ask High-Level Questions First (3 minutes)

[SCREEN: Claude Code interface]

Professor:

"Here's a critical principle: Start broad, then go deep. Don't immediately dive into specific functions or files. First, understand the forest, THEN the trees.

Let me show you the questions I typically ask when exploring a new codebase. These are listed in the order I recommend asking them.

Question 1: What does this application do?"

[TYPE in Claude Code]:

```
What is the purpose of this application? What problem does it solve and
what are its main features?
```

[SCREEN: Shows Claude's response]

Professor:

"Excellent! Claude tells us this is a flashcard learning application with:

- Deck management
- Card creation
- Spaced repetition for studying
- Progress tracking

Now I understand WHAT this app is for. Next question:

Question 2: What's the technical architecture?"

[TYPE in Claude Code]:

```
Explain the technical architecture of this application. What are the major
components and how do they interact?
```

[SCREEN: Shows Claude explaining the architecture]

Professor:

"Great! Now I know:

- React frontend communicates with FastAPI backend via REST API
- PostgreSQL database stores user data
- SQLModel ORM handles database operations
- React Query manages server state on the frontend
- JWT authentication (though simplified in this version)

I'm building a mental model of the system. Next:

Question 3: How is the code organized?"

[TYPE in Claude Code]:

```
Walk me through the project structure. What is the purpose of each major
directory in both frontend and backend?
```

[SCREEN: Shows Claude's detailed breakdown]

Professor:

"Perfect! Claude explains:

Backend structure:

- `app/api/routes/` - API endpoint definitions
- `app/models/` - Database models
- `app/schemas/` - Request/response validation
- `app/services/` - Business logic
- `tests/` - Test suite

Frontend structure:

- `src/pages/` - Main page components
- `src/components/` - Reusable UI components
- `src/lib/` - Utilities and API client
- `src/types/` - TypeScript type definitions

This is the architectural layer cake. I now know where to find things.

Question 4: What are the key data models?"

[TYPE in Claude Code]:

```
What are the main database models in this application and how do they
relate to each other?
```

[SCREEN: Shows Claude explaining User, Deck, Card, and study-related models]

Professor:

"Excellent! Now I understand the data model:

- Users own Decks
- Decks contain Cards
- Users have SRSReviews for Cards (spaced repetition state)
- QuizSessions track study sessions with QuizResponses for individual cards

This is the backbone of the application. With these four questions, I now have a solid high-level understanding:

1. WHAT the app does
2. HOW it's architected
3. WHERE the code is organized
4. WHAT data it manages

Now we can go deeper with specific questions."

PART 3: Ask Detailed Questions from High-Level Answers (3 minutes)

[SCREEN: Claude Code interface]

Professor:

"Now that we understand the high level, we can ask targeted questions about specific parts that interest us. Let's say I want to understand the spaced repetition system - that's a key feature.

From our high-level questions, I learned that:

- There's an SRSReview model
- There's study session functionality
- It's implemented somewhere in the services layer

Let me dig deeper:

Detailed Question 1: How does spaced repetition work?"

[TYPE in Claude Code]:

```
I see this app uses spaced repetition (SRS). Can you explain how the SM-2
algorithm is implemented? Show me the relevant code in
backend/app/services/study.py
```

[SCREEN: Shows Claude opening study.py and explaining the _apply_sm2 function]

Professor:

"Fantastic! Claude shows me the `_apply_sm2()` function and explains:

- Quality ratings from 0-5 determine the next review interval
- Quality < 3 resets the card (you got it wrong)
- Quality ≥ 3 increases the interval exponentially
- An 'easiness' factor adjusts how quickly intervals grow

I can see the actual implementation at backend/app/services/study.py:67. Let me ask a follow-up:

Follow-up Question:"

[TYPE in Claude Code]:

```
When a user studies a card, what's the complete flow from the frontend
through to updating the SRS state? Walk me through the code path.
```

[SCREEN: Shows Claude tracing through multiple files]

Professor:

"Look at this! Claude traces the entire flow:

1. Frontend: User clicks quality rating in `StudySessionPage.tsx`
2. Frontend: API call to `POST /api/v1/study/sessions/{id}/answer`
3. Backend: Route handler in `routes/study.py` receives request
4. Backend: Calls `study_service.record_answer()`
5. Backend: `record_answer()` saves QuizResponse
6. Backend: If in review mode, calls `_get_review_state()` and `_apply_sm2()`
7. Backend: Updates SRSReview model with new due date
8. Backend: Returns response to frontend

This is GOLD. I now understand exactly how a core feature works, across the entire stack.

Let me ask another detailed question about something else:

Detailed Question 2: Authentication flow"

[TYPE in Claude Code]:

```
I noticed in CLAUDE.md that authentication is simplified in this version.
How does it actually work? Show me the code in backend/app/api/deps.py
```

[SCREEN: Shows Claude explaining the simplified auth]

Professor:

"Ah, interesting! Claude shows me that this version has simplified authentication:

- There's a `get_or_create_default_user()` function
- It automatically creates a single user if none exists
- All requests use this default user
- No real login/signup validation

This is intentionally simplified for learning purposes. The full version has proper JWT authentication.

This is important to know! If I tried to implement a feature that relied on multiple users, it wouldn't work in this version.

One more detailed question:"

[TYPE in Claude Code]:

```
How does the frontend manage API requests? Show me the Axios configuration
and how it handles authentication.
```

[SCREEN: Shows Claude opening frontend/src/lib/apiClient.ts]

Professor:

"Claude shows me the API client setup:

- Base URL configured from environment variables
- Axios interceptors for request/response handling
- Simplified auth headers

Now I understand how frontend and backend communicate. Notice how each detailed question builds on what we learned from the high-level questions? This is the systematic approach:

1. Understand the big picture
2. Identify areas of interest
3. Drill down with specific questions
4. Follow the code flow through multiple files"

PART 4: Verify Claude's Answers (2 minutes)

[SCREEN: Split screen - Claude Code on left, VS Code on right]

Professor:

"Now here's a critical skill: ALWAYS verify what Claude tells you. Claude is incredibly helpful, but it's not infallible. It might miss details or occasionally misinterpret code.

Let me show you how to verify. Claude told us the SM-2 algorithm is in `study.py` at the `_apply_sm2()` function. Let's check:"

[SCREEN: Open backend/app/services/study.py in VS Code, navigate to line 67]

Professor:

"Line 67, there it is! The `_apply_sm2()` function. Let me read through it quickly..."

[SCREEN: Scroll through the function]

Professor:

"Yes, this matches what Claude explained:

- Quality check (line 68-69)
- Reset if quality < 3 (lines 71-73)
- Increase interval based on repetitions (lines 75-83)
- Update easiness factor (lines 85-88)
- Set new due date (line 90)

Claude's explanation was accurate. But let me show you something important - asking Claude to explain WHY:"

[TYPE in Claude Code]:

```
In the _apply_sm2 function, why does it use max(1.3, ...) when calculating
the new easiness factor? What would happen if we didn't have that minimum?
```

[SCREEN: Shows Claude's explanation]

Professor:

"Claude explains that 1.3 is the minimum easiness factor to prevent the interval from shrinking too much if a user consistently rates cards low quality. Without it, the easiness could become very small or even negative, breaking the algorithm.

This is verification through understanding. Don't just accept that code works - understand WHY it's written that way.

Another verification technique: Ask Claude to show you tests:"

[TYPE in Claude Code]:

```
Are there tests for the SM-2 algorithm? Show me how it's tested.
```

[SCREEN: Shows Claude opening backend/tests/test_srs.py]

Professor:

"Perfect! There ARE tests. Let me verify by running them:"

[SCREEN: Terminal]

```
cd backend
DATABASE_URL="sqlite:///./test.db" pytest tests/test_srs.py -v
```

[SCREEN: Shows tests passing]

Professor:

"All tests pass! This verifies that:

1. The code exists where Claude said it does
2. It behaves as Claude explained
3. The implementation is tested and working

This is how you verify: Look at the actual code, understand the logic, and run the tests."

PART 5: Building Your Understanding Systematically (2 minutes)

[SCREEN: Whiteboard or diagram software]

Professor:

"Let me show you how to document what you've learned. I like to create a simple diagram or notes as I explore. Let me create a quick reference:"

[SCREEN: Create a simple text file or diagram]

```
MINIMAL-FLASHCARDS UNDERSTANDING MAP

ARCHITECTURE:
- Frontend: React + TypeScript (port 5173)
- Backend: FastAPI + Python (port 8000)
- Database: PostgreSQL
- Communication: REST API

DATA MODEL:
User --(owns)--> Deck --(contains)--> Card
User + Card --> SRSReview (spaced repetition state)
User + Deck --> QuizSession --> QuizResponse

KEY FLOWS:
1. Study Flow:
   Frontend (StudySessionPage)
   → POST /api/v1/study/sessions/{id}/answer
   → routes/study.py
   → services/study.py (record_answer)
   → _apply_sm2() updates SRSReview

2. Deck Management:
   Frontend (DashboardPage/DeckDetailPage)
   → routes/decks.py
   → services/decks.py
   → models/deck.py & models/card.py

SPECIAL NOTES:
- Auth is simplified (single default user)
- Only basic card types in this version
- SM-2 algorithm: quality 0-5, interval grows exponentially
- Frontend uses React Query for state management
```

Professor:

"This is MY understanding map. Yours might look different - maybe you prefer visual diagrams or different organization. The point is: document what you learn.

Now, let me give you a suggested question flow for ANY new codebase:

Phase 1: Big Picture (5-10 minutes)

1. What does this application do?
2. What's the technical architecture?
3. How is the code organized?
4. What are the key data models?

Phase 2: Deep Dive (20-30 minutes) 5. Pick a key feature: How is it implemented end-to-end? 6. Pick another feature: Trace the code flow 7. How is authentication/authorization handled? 8. How is error

handling done? 9. What's the testing strategy?

Phase 3: Verification (10-15 minutes) 10. Read the actual code for key functions 11. Run the tests 12. Try running the application 13. Make a small change and see what breaks

This systematic approach means you can understand a complex codebase in under an hour, versus days of random file browsing."

CLOSING (1 minute)

[SCREEN: Back to Claude Code interface]

Professor:

"Let's recap what we learned today:

The Systematic Approach:

1. **Clone and Initialize** - Run /init to let Claude analyze the codebase
2. **Start Broad** - Ask high-level questions about purpose, architecture, and organization
3. **Go Deep** - Follow up with specific questions about features that interest you
4. **Verify Everything** - Look at actual code, understand the why, run tests
5. **Document Your Understanding** - Create a personal reference map

Key Questions to Always Ask:

- What does this do?
- How is it architected?
- Where is the code organized?
- What data does it manage?
- How does [specific feature] work?

Remember: Claude is your guide, not your crutch. The goal is to UNDERSTAND the codebase, not just have Claude tell you about it. Verify, test, and build your own mental model.

In our next tutorial, we'll use everything we've learned to plan and implement a new feature: adding multiple-choice questions to our flashcard app. We'll see how Claude can help us design the feature, identify all the changes needed, and implement it correctly.

See you next time!"

INSTRUCTOR NOTES:

Timing Breakdown:

- Introduction: 1 min
- Part 1 (Clone & Init): 2 min
- Part 2 (High-Level Questions): 3 min
- Part 3 (Detailed Questions): 3 min
- Part 4 (Verification): 2 min

- Part 5 (Systematic Approach): 2 min
- Closing: 1 min
- **Total: ~12 minutes**

Key Points to Emphasize:

1. Always run /init first - it's worth the 30-60 seconds
2. Broad before deep - understand the forest before the trees
3. Follow the code flow across multiple files
4. ALWAYS verify Claude's explanations
5. Document your understanding as you go

Preparation Needed:

- Fresh clone of the repository in a clean directory
- Backend tests ready to run
- Have diagram/whiteboard tool ready for visual learners
- Prepare the "understanding map" template beforehand

Questions Students Often Ask:

- Q: "How many questions should I ask?" A: As many as you need to feel comfortable. For a small project, 5-10. For a large one, could be 20-30 over several sessions.
- Q: "What if Claude gives me wrong information?" A: That's why we verify! Look at the code, run tests, try the feature. Trust but verify.
- Q: "Should I read every file Claude mentions?" A: Start with the key files Claude highlights. You can always go deeper later.

Common Pitfalls to Address:

- Don't skip the high-level questions - students want to jump straight to code
- Don't accept Claude's answers blindly - emphasize verification
- Don't try to understand everything at once - focus on what's relevant to your goals

Live Coding Moments:

- Actually run the tests to show verification
- Open the actual files to show matching Claude's descriptions
- Make a small change to demonstrate understanding (optional, time permitting)