🎓 3. `@@ERROR` System Function

✅ What is `@@ERROR`?

Definition:
`@@ERROR` is a **system function** in SQL Server that returns the **error number** of the **most recently executed SQL statement**.

- If the last statement ran **successfully**, `@@ERROR` returns `0`.
- If the statement **failed**, `@@ERROR` returns a **non-zero error code**.

Teaching Script:

"Think of `@@ERROR` as SQL Server's way of saying: 'Did the last thing I did go wrong?' If so, it gives you a code number to tell what went wrong. If not, it gives you a zero."

⚠️ Behavior and Limitations

✅ Behavior:

- Works **only for the immediately preceding SQL statement**.
- If you run another statement (even a `PRINT` or `SELECT`), `@@ERROR` resets to 0.
- Works in **basic error checking** scenarios.

❌ Limitations:

- Doesn't automatically handle errors — you must manually check and act.
- Very **easy to miss** the correct value if not checked immediately.
- Only returns the **error number** — not the message, line, or procedure.
- Not useful for **grouping** error handling logic in complex scripts.

Teaching Script:

"One major weakness of `@@ERROR` is that it has a short memory — it forgets the error as soon as something else runs. That's why you must check it immediately, or the information is lost."

🧠 Capturing `@@ERROR` After Statements

To use `@@ERROR`, you must:

1. Execute your SQL statement.
2. Immediately check `@@ERROR` and store it if needed.

Example pattern (no code here, just concept):

```
pgsql
CopyEdit
Run a statement
Check @@ERROR right after
Decide what to do based on its value
```

Teaching Script:

"If you want to check whether a line failed, you must ask SQL Server *right after it happens*. If you wait even one more command, `@@ERROR` goes back to 0 — even if the last real command failed."

📝 Why It Must Be Checked Immediately

- `@@ERROR` is **cleared and reset** after every SQL statement.
- Even a harmless command like `PRINT` or `SELECT` will overwrite it.

Bad Practice:
Wait too long → `@@ERROR` returns 0 even after failure.

Good Practice:
Store `@@ERROR` in a variable **immediately**, then use it later if needed.

Teaching Script:

"You have to think of `@@ERROR` like a post-it note that SQL gives you after every command. If you don't read it immediately and write it down, the next command will replace it — and the error info is gone forever."

🧾 Summary Recap

Concept	Description
What it is	A system function that shows the last error
Success return value	0
Failure return value	Non-zero error number
Limitation	Must be checked immediately after a statement
Doesn't provide	Message, line number, or details

Ready to move on to **TRY...CATCH**, or would you like to add an exercise or discussion here first?

🎓 4. TRY...CATCH Block in SQL Server

✅ Introduction to Structured Error Handling

Definition:
The `TRY...CATCH` block is SQL Server's structured way of handling errors — similar to `try-catch` in many programming languages (like C#, Java, or Python).

Purpose:

- Allows you to write cleaner, more readable, and more reliable code.
- Automatically **catches errors** inside a `TRY` block and sends control to a `CATCH` block.

Teaching Script:

"With `TRY...CATCH`, we can finally organize our error-handling logic cleanly — like wrapping your risky code in a safety net. If something fails, SQL Server won't panic — it just jumps to the CATCH block, where we can deal with the problem."

🧱 Syntax and Structure

```
sql
CopyEdit
BEGIN TRY
    -- Code that might fail
END TRY
BEGIN CATCH
    -- Code to handle the error
END CATCH
```

Key Points:

- Code inside `BEGIN TRY` is executed normally.
- If **no error** occurs, the `CATCH` block is skipped.
- If **an error** occurs, control jumps immediately to `BEGIN CATCH`.

Teaching Script:

"Think of TRY as the zone where you're doing something risky. If anything breaks in there, SQL Server skips the rest of TRY and lands in the CATCH block — like an emergency exit."

🧩 TRY Block

- Put the code you want to **monitor for errors** here.
- If all goes well, CATCH is ignored.
- If anything inside fails, the rest of the TRY block is **skipped**, and control moves to CATCH.

Teaching Tip:
Use real-world language — "Try this risky thing..."

🧲 CATCH Block

- Automatically triggered **only if an error occurs** in the TRY block.
- You can use special functions here to learn about the error:
  - `ERROR_MESSAGE()`
  - `ERROR_NUMBER()`
  - `ERROR_LINE()`
    (You'll teach these in the next section.)
- You can log the error, return a message, or even re-throw it using `THROW`.

Teaching Script:

"This is where you decide how to respond. Maybe just show a message. Maybe log it. Or maybe stop everything. It's up to you."

📦 Scope of TRY...CATCH

- It works at the **statement level**:
  - One failing statement inside TRY will send control to CATCH.
- Only **run-time errors** are caught — not compile-time errors.
- Each TRY...CATCH is **independent** — you can nest them for more control (teach this later when students know more).

Teaching Script:

"TRY...CATCH watches for mistakes that happen *while* SQL runs. It won't help if your code has typos or is missing objects — that's a compile-time issue, not a run-time one."

🔄 What Happens When an Error is Caught

- Execution inside TRY **stops** at the error.
- SQL Server jumps to CATCH.
- Code in CATCH runs — you can handle, log, or display error info.
- The code **after** the CATCH block **will still run**, unless explicitly stopped.

Teaching Script:

"As soon as SQL hits a problem in TRY, it abandons ship and runs whatever you've written in the CATCH block — but it won't crash the whole system like before."

⚖️ Differences from `@@ERROR`

Feature	`@@ERROR`	TRY...CATCH
Manual check needed	Yes (immediately after statement)	No (automatically jumps to CATCH)
Code structure	Scattered	Structured
Info available	Error number only	Rich info (message, line, procedure)
Catches all run-time errors	Limited	Catches most run-time errors
Works with batches	Weakly	Much more reliable
Suitable for modern use	No	Yes (preferred method)

Teaching Script:

"`@@ERROR` is like checking every time something might go wrong. TRY...CATCH is like installing a fire alarm — it alerts you automatically and sends you to the emergency plan."

📝 Summary Recap

- Use `TRY...CATCH` to handle **run-time** errors automatically.
- Place risky code in `TRY`, and fallback logic in `CATCH`.
- It makes your SQL more reliable and readable.
- Preferred over `@@ERROR` for all modern SQL development.

🧠 What Are ERROR Functions?

Definition:
SQL Server provides a set of **built-in functions** that you can only use **inside a CATCH block** to get detailed information about the error that was caught.

These functions give you the **context of the error**, making debugging and error logging much easier.

📋 List of Key ERROR Functions

Function	What It Returns
`ERROR_NUMBER()`	The error number (like a code)
`ERROR_MESSAGE()`	The full text description of the error
`ERROR_SEVERITY()`	The severity level of the error (1 to 25)
`ERROR_STATE()`	A code for internal tracking (not usually user-facing)
`ERROR_LINE()`	The line number inside the batch or procedure where it failed
`ERROR_PROCEDURE()`	The name of the stored procedure where the error occurred

🧾 Detailed Explanation of Each Function

1️⃣ `ERROR_NUMBER()`

- Returns the **numeric error code**.
- Helps identify the type of error (e.g., 547 = foreign key violation).

Teaching Tip:
Use this for error classification or conditional handling.

“This is like the unique ID number for the error. It helps you know *what* went wrong.”

2️⃣ `ERROR_MESSAGE()`

- Returns the **full text** of the error message.
- Useful for logging or showing to the user/admin.

“This is the human-readable version — the part you'd show in an error log or report.”

3️⃣ `ERROR_SEVERITY()`

- Tells you **how serious** the error is:
    - 0–10: Informational
    - 11–16: User errors
    - 17–25: System errors

"This tells you whether it's a minor issue or a serious problem you need to escalate."

4️⃣ `ERROR_STATE()`

- Used internally by SQL Server to distinguish between different causes of the same error number.
- Less commonly used by developers.

"Think of this like an extra detail or sub-code. It's not always meaningful, but it can help with complex debugging."

5️⃣ `ERROR_LINE()`

- Tells you **exactly which line** of SQL caused the error.

"This makes it easy to go back and fix the line that broke — a real time-saver for debugging."

6️⃣ `ERROR_PROCEDURE()`

- If the error happened inside a **stored procedure**, this returns its name.
- If not, it returns `NULL`.

"If the error came from inside a stored procedure, this tells you which one. If it's `NULL`, the error happened in ad-hoc SQL."

🧪 How and Where to Use Them

- These functions can **only be used inside the CATCH block**.
- Use them to:
    - **Log error details** to a table or file.
    - **Display user-friendly messages**.
    - **Re-throw errors** with more info using `THROW`.

🧭 Why These Functions Matter

- Provide **rich, precise information** for developers and support teams.
- Improve **error logging and monitoring** in real-world systems.
- Allow better decision-making in CATCH blocks (e.g., continue, stop, notify).

Teaching Script:

"Without these functions, we'd just know something went wrong — but not what, where, or why. These tools let us pinpoint the issue like a magnifying glass."

📝 Summary Recap

Function	Purpose
`ERROR_NUMBER()`	What error happened
`ERROR_MESSAGE()`	What it said
`ERROR_SEVERITY()`	How bad it was
`ERROR_STATE()`	Extra detail
`ERROR_LINE()`	Where it happened
`ERROR_PROCEDURE()`	Which procedure it happened in

🎓 6. THROW Statement in SQL Server

✅ What is the `THROW` Statement?

Definition:
`THROW` is a SQL Server command used to:

1. **Raise a new error**, or
2. **Re-throw the original error** inside a CATCH block.

It is the **modern replacement** for the older `RAISERROR` command (which you'll teach in the next section).

🪓 Why Use `THROW`?

- Simpler and cleaner syntax
- Automatically captures current error context (when re-throwing)

- Always terminates the batch or transfers control
- Works well inside TRY...CATCH blocks
- Fully supports modern error-handling logic

Teaching Script:

"If TRY...CATCH is how you catch the problem, then THROW is how you report it back or pass it on. It's like saying, 'I caught the error, but now I'm sending it up the chain.'"

📐 Syntax of THROW

1️⃣ Re-throw the current error:

```
sql
CopyEdit
THROW;
```

- Used inside a CATCH block.
- Passes the original error unchanged.

2️⃣ Throw a custom error:

```
sql
CopyEdit
THROW error_number, 'message', state;
```

- `error_number`: Integer > 50000 (custom-defined)
- `message`: Text (up to 2048 characters)
- `state`: Integer from 0 to 255 (used to differentiate situations)

📌 When to Use Each Form

Purpose	Syntax	Use Case
Re-throw original error	`THROW;`	After logging the error inside CATCH
Raise custom error	`THROW num, msg, state;`	When enforcing business rules, input checks, etc.

Teaching Script:

"Sometimes you want to log the error and move on. Other times, you want to stop execution and tell the caller something went wrong — THROW helps you do both."

⚠️ Rules and Notes

- `THROW` **must** be used:
  - Inside a **CATCH block** (for re-throwing), or
  - As a standalone statement (for new errors)
- Error numbers **below 50000** are reserved by SQL Server
- Always stops execution unless caught by another TRY...CATCH

🔁 THROW vs RAISERROR (Preview)

Feature	`THROW`	`RAISERROR`
Modern SQL support	Yes (SQL Server 2012+)	Older compatibility
Re-throwing	Simple and built-in	Not supported
Syntax	Clean and simple	Complex, outdated
Preferred method	✅	❌ (legacy use only)

"In modern SQL, always prefer THROW. We'll cover RAISERROR next — mostly to understand legacy code or systems that still use it."

📝 Summary Recap

- `THROW` is the **preferred way** to raise or re-raise errors in SQL Server.
- Use `THROW;` to re-throw errors caught in CATCH blocks.
- Use `THROW error_number, message, state;` to raise **custom** errors.
- Must use custom error numbers **above 50000**.

🎓 7. `RAISERROR` (Legacy)

✅ What is `RAISERROR`?

Definition:
`RAISERROR` is an older SQL Server statement used to **generate errors manually** during the execution of T-SQL code.

- It was the **primary tool for error generation** before `THROW` was introduced in SQL Server 2012.
- Still supported for **backward compatibility**, but not recommended for new development.

Teaching Script:

"Before SQL Server 2012, `RAISERROR` was the only way to raise custom errors. Today, we mostly use it when working with legacy systems, but it's important to understand how it works."

🧱 Basic Syntax

```
sql
CopyEdit
RAISERROR ('Message text', severity, state);
```

- `'Message text'`: Custom error message (up to 400 characters)
- `severity`: Integer from 0 to 25 (level of seriousness)
- `state`: A value from 0 to 255 (used for identifying different sources or contexts)

📌 Key Concepts

🧨 Severity Levels

- 0–10: Informational messages
- 11–16: User-defined errors (can be caught)
- 17–19: More severe (still user level, but harder to handle)
- 20–25: System errors (terminate connection/session)

"Severity levels let you signal how serious the problem is — from a warning to a full system crash."

🧾 State

- Developer-defined number to help identify the source of the error.
- Example use: differentiate between types of errors under the same error number.

🧠 Extended Syntax (with variables or formatting)

```
sql
CopyEdit
RAISERROR ('Value %d is not valid', 16, 1, @Value);
```

- You can insert variables using format specifiers like `%d`, `%s`, etc.
- Similar to `printf()` in C-style languages.

🔒 Error Number Rules

- If using a **custom error number**, it must be defined in `sys.messages` with `sp_addmessage`.
- Otherwise, use message text directly in the command.

🔁 RAISERROR vs THROW (Comparison)

Feature	`RAISERROR`	`THROW`
Syntax simplicity	More complex	Cleaner and modern
Re-throw support	❌ Cannot re-throw original error	✅ Yes
Message length limit	400 characters	2048 characters
Formatting support	✅ With placeholders	❌ No formatting
Custom error number	Requires `sp_addmessage`	Can define directly in statement
Preferred for new code	❌ Legacy only	✅ Yes

⚠️ Limitations of `RAISERROR`

- More complex to use.
- Cannot re-throw original errors (must create new ones).
- Requires registration of permanent messages if not using plain text.
- Doesn't integrate as cleanly with TRY...CATCH as `THROW`.

Teaching Script:

"While `RAISERROR` gives you flexibility with formatting and error numbers, it's also more effort and more prone to errors. That's why SQL Server moved to `THROW` — simpler, safer, and more reliable."

📝 Summary Recap

- `RAISERROR` is used to raise errors manually (mostly in legacy code).
- It allows for **custom messages, severity levels, and formatting**.
- It's more complex and less powerful than `THROW`.
- Best used **only in existing systems** that already rely on it.

🧱 What Are Nested TRY...CATCH Blocks?

Definition:
Nested TRY...CATCH blocks are **TRY...CATCH statements placed inside each other** to handle errors **at different levels** of your code logic.

🧭 Why Nest TRY...CATCH?

- Handle **specific errors locally** while letting more serious ones bubble up.
- Add **extra control** — you might retry, log, or exit based on error type.
- Structure your logic like layered error protection.

Teaching Script:

"Think of nested TRY...CATCH like safety layers in a building. If something fails on the top floor, you might have a localized alarm — but a bigger failure might trigger the whole building's response system."

📐 Basic Structure of Nested TRY...CATCH

```
sql
CopyEdit
BEGIN TRY
    -- Outer TRY
    BEGIN TRY
        -- Inner TRY (specific operation)
    END TRY
    BEGIN CATCH
        -- Inner CATCH (handle minor/local error)
    END CATCH
END TRY
BEGIN CATCH
    -- Outer CATCH (handle more serious error)
END CATCH
```

Key Points:

- Each TRY block must have its own CATCH.

- You can nest as deep as needed (but keep it readable).
- Inner CATCH can **handle and suppress** an error, or **re-throw** it to the outer CATCH using `THROW`.

🧠 When to Use Nested TRY...CATCH

Situation	What You Do
Minor error you want to log only	Handle in inner CATCH, don't re-throw
Major error or failure to recover	Re-throw in inner CATCH, handle in outer CATCH
Isolated step might fail	Wrap it in its own TRY...CATCH

Teaching Script:

"Sometimes you just want to log a harmless issue and keep going. Other times, you want to stop the process. Nesting TRY...CATCH gives you the power to decide how to handle different kinds of errors at different levels."

💡 Light Introduction to Transactions (Without Deep Dive)

Since your students haven't learned transactions yet, introduce only the **concept**:

What is a Transaction?
A transaction is a group of operations that must **all succeed or all fail together** — like a "bundle."

"Imagine transferring money between two accounts. You need to subtract from one and add to the other. If one step fails, the whole thing should cancel — that's what transactions are for."

⚠️ Why Error Handling Matters with Transactions

- If an error occurs during a transaction, you must decide whether to **cancel (rollback)** or **continue (commit)**.
- If handled badly, part of the data might be saved, and part might not — causing **data inconsistency**.

"This is why structured error handling — especially with transactions — is so important. It prevents your system from getting into an unpredictable or broken state."

⛳ Optional Mention: `XACT_STATE()` (Just Naming It)

If you want to **name-drop** a concept for later:

- `XACT_STATE()` is a function used to **check the state of a transaction** in error handling.
- You'll teach this in more detail when you cover transactions.

📝 Summary Recap

Concept	Summary
Nested TRY...CATCH	Error handling within error handling
Purpose	Isolate, log, or escalate errors as needed
Use case	Layered logic, different responses based on error type
Transaction tie-in	Needed when you group changes that must succeed or fail together

🎓 9. Best Practices for Error Handling in SQL Server

🧠 1. Always Use `TRY...CATCH` for Run-Time Error Handling

- Structured and reliable.
- Automatically catches errors without manual checks.
- Preferred in all modern SQL Server development.

Teaching Script:

"The first and most important rule — always use TRY...CATCH. It gives you clean, readable, and safe error handling out of the box."

📌 2. Avoid `@@ERROR` in New Code

- It's old, error-prone, and easy to misuse.
- Use it only if you're maintaining legacy systems.

"Think of `@@ERROR` like a landline phone — it still works, but we've got better tools now."

📋 3. Use `THROW` Instead of `RAISERROR` (For New Code)

- `THROW` is cleaner, more modern, and integrates with TRY...CATCH.
- Only use `RAISERROR` if you're dealing with older systems or need formatting features.

🧾 4. Capture and Log Error Details

- Use `ERROR_MESSAGE()`, `ERROR_LINE()`, etc., to capture what happened.
- Save this info in an **error log table** for debugging and auditing.

"Don't just handle the error — record it. Future-you or your support team will thank you."

🚧 5. Design for Failures (Graceful Degradation)

- Not every error needs to stop everything.
- Decide which errors can be logged and skipped vs. those that require halting.

"Your system should bend without breaking. Handle small errors quietly; escalate the big ones."

🔍 6. Be Specific in the CATCH Block

- Don't just say "an error occurred" — capture **what**, **where**, and **why**.
- Combine `ERROR_MESSAGE()` with `ERROR_LINE()` and `ERROR_PROCEDURE()` to give full context.

⛓️ 7. Keep TRY Blocks Short and Focused

- Easier to isolate and debug issues.
- Better visibility into what caused the error.

"If your TRY block is 100 lines long, it's hard to know what failed. Keep it tight and meaningful."

🔁 8. Don't Swallow Errors Silently

- Avoid writing empty CATCH blocks (e.g., `BEGIN CATCH END`).
- Always log, raise, or notify somehow — even for non-critical errors.

"Imagine your car showing no warning lights even if the engine fails — don't do that in your SQL."

📊 9. Plan for Transactions in Error Handling (When Ready)

- Once they learn transactions, emphasize:
    - Roll back if an error occurs.

    - Check `XACT_STATE()` before continuing.
- Combine TRY...CATCH with transaction control for full safety.

🧭 10. Test Error Handling Intentionally

- Simulate errors to ensure your handling logic works as expected.
- Use invalid data, `THROW`, or fake failures during development/testing.

“Don’t wait for a real crash to find out your error handling doesn’t work. Test it deliberately.”

✅ Summary Recap of Best Practices

Practice	Why It Matters
Use TRY...CATCH	Modern, structured error control
Prefer THROW	Cleaner than RAISERROR
Log all errors	Helps debugging and maintenance
Be clear and specific	Makes your system easier to support
Keep TRY short	Easier debugging
Never swallow errors silently	Problems should never disappear quietly

Let me know if you'd like:

- Review slides
- Exercises/quiz for students
- Error-handling use case scenarios
- Or move on to **examples** as you mentioned earlier

Ready when you are!