ARITHMETIC IN RUST

- This video covers arithmetic operations in Rust with different data types, type conversions, and casting.
- Focus is on nuances in a statically typed language, especially overflows.

BASIC VARIABLES AND DATA TYPES

- Example: `let x: u8 = 9;` and `let y: i8 = 10;`
- Comments in Rust: `// This is a comment.`
- Range for `u8`: 0 to 255.
- Range for `i8`: -128 to 127.
- Assigning a value outside the type's range results in a compile-time error.
- A "literal" is a primitive value typed directly into the code (e.g., 256, 1, 1.23).

ADDITION

- Adding variables of different types (e.g., `u8 + i8`) results in a compile-time error.
- Types must be the same to be added without conversion.

FLOATING-POINT TYPES

- Similar type restrictions apply to floating-point types (`f32`, `f64`).
- Integer values cannot be directly assigned to float types; use float literals (e.g., `10.0` for `f32`).

OVERFLOWS

- Adding values that exceed the maximum value for a type causes an overflow error.
- Example: `let x: u8 = 255; let y: u8 = 1; let z = x + y;` results in an overflow error.
- To fix, cast to a larger type (e.g., `u16`).
- Subtracting can also cause underflow errors with unsigned types.

DIVISION

- Result of division is the same type as the operands.
- Integer division truncates the decimal portion.
- To get a floating-point result, use floating-point types for the operands.

MULTIPLICATION

- Multiplication uses the `*` operator.
- Operands of type `f64` will return a variable of type `f64`.

MODULUS OPERATOR

- The modulus operator (`%`) returns the remainder of a division.
- Example: `x % y`

TYPE CONVERSIONS AND CASTING

- Writing Literals as a Specific Type:
- Append the type directly to the number (e.g., `255.0f32`).
- Use an underscore to specify they type (e.g. `255_i8`)
- Use underscores to improve readability (e.g., `127_000_i64`).
- Using `as` Keyword:
- `value as type` (e.g., `x as i64`).

EXPLICIT TYPE CONVERSION

- Rust does not automatically convert types; explicit conversion is required.
- Use the `as` keyword to convert between types.
- Convert smaller types to larger types to prevent potential overflows.
- Converting larger types to smaller types can lead to unexpected behavior due to two's complement wrapping.

STRING TO NUMBER CONVERSION

- Demonstrates converting user input (string) to an integer.

- Code snippet includes:
- `use std::io;`
- Creating a mutable string: `let mut input = String::new();`
- Reading user input: `std::io::stdin().read_line(&mut input).expect("Expected to read line");`
- Trimming whitespace: `input.trim()` to remove the newline character.

- Parsing the string: `input.trim().parse::<i64>().unwrap();`
- `parse::<i64>()` attempts to convert the string to an `i64`.
- `.unwrap()` extracts the parsed value or panics if parsing fails.
- Example: `let int_input: i64 = input.trim().parse::<i64>().unwrap();`
- Error handling: If the input is not a valid integer, `unwrap()` will cause a runtime exception.