ARITHMETIC IN RUST

- This video covers arithmetic in Rust, type conversions, and casting.
- Includes information on overflows.

BASIC VARIABLES AND DATA TYPES

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

ADDING VALUES OF DIFFERENT TYPES

- Rust does not allow direct arithmetic operations between different data types.
- Example: Cannot add `u8` and `i8` without conversion.
- Same applies to floating-point types like `f32` and `f64`.
- Need to use `2.0f32` rather than `2` when assigning to a floating point variable.

DIVISION, SUBTRACTION & OVERFLOWS

- Adding values beyond the maximum value of a type leads to an overflow error, preventing compilation.
- Subtraction can also lead to overflow errors if the result is negative for an unsigned integer type.

DIVISION RESULTS

- Integer division truncates the decimal portion.
- The result of an arithmetic operation has the same data type as the operands.
- Example: Dividing `u8` by `u8` will result in a `u8`.
- Floating-point division produces a floating-point result.

MULTIPLICATION AND MODULUS

- Multiplication uses the asterisk `*` operator.
- Modulus operator `%` returns the remainder of a division.

TYPE CONVERSIONS AND CASTING

- Literals can be specified with a type suffix, e.g., `255_f32` or `10_i8`.
- Can use underscores to improve the readability of large numbers e.g. `127_000`.

- The `as` keyword is used for explicit type conversion or casting.
- Example: `x as i64` converts `x` to an `i64`.
- Smaller types should be cast to larger types to prevent loss of information.
- Converting larger types to smaller types can cause unexpected overflow behavior.

STRING TO NUMBER CONVERSION

- Example of getting user input and converting it to a number:
- `use std::io;`
- `let mut input = String::new();`
- `std::io::stdin().read_line(&mut input).expect("...");`
- `let int_input: i64 = input.trim().parse().unwrap();`
- `.trim()` removes the newline character from the input.
- `.parse()` converts the string to a number.
- `.unwrap()` gets the value from the `Result`.
- Ensure user input is a valid number to avoid runtime exceptions when using `unwrap()`.