# Arithmatic and Type Casting

## Arithmatic

- **Note Overflows** careful with overflows during arithmatic operations
    - Examples:

```
let x: u8 = 255;
let y: u8 = 1;
let z = x + y;// error OVERFLOW, 256 is greater than 255
let z = y - x; // error OVERFLOW, uint cannot be signed
println("{}", z);
```

- You cannot add two different types together
    - Examples:

```
    let x: u8 = 12;
    let y: i8 = 10;
    let z = x + y;
    println("{}", z); // error cannot add two different types
```

- Truncation
    - If an integer needs to take in a decimal value, it will automatically truncate (round down)
    - Examples:

```
let x: u8 = 255;
let y: u8 = 10;
let z = x / y;
println("{}", z); // z = 25, the 25.5 gets truncated
```

## Type Casting

- There are two ways to cast
- One way is to append the type
    - `let x = 20i8;//without underscore`
    - `let y = 20_i8;//with underscore`
    - The underscore method can be used to write numbers in a more clear way
        - let x = 200_000_i64; // this is equal to 200,000

- The other way is to use the `as` key word
    - `let x = 200_000 as i64;`
    - Examples:

```
let x = 200_000 as i64;
let y = 200 as i32;
let z = x / (y as i64); // converts type before arithmetic
```

- You can check the max of each type using the MAX key word
    - `let x = i32::MAX;`

## String to Integer conversion

- Refreshing how to read in a string

```
let mut input = String::new();
io::stdin().read_line(&mut input).expect("expected string");
```

- To convert into a an integer you first need to trim the string
    - Trimming the string removes the new line character
    - `input.trim()`
- Then we need to parse() to return a result, it'll parse to see if it can be parsed into an integer
    - `input.trim().parse()`
- Last part we need to unwrap the parse to the actual integer type
- `input.trim().parse().unwrap();`
- We need to give an explicit type conversion for the parse
- `let x: i32 = input.trim().parse().unwrap();`