1 - let variable_name = value (variables are by default immutable)

```
let variable = 24;
```

2 - adding a mut keyword during declaration

```
let mut variable2 = 21;
```

3 -

```
let val  = 14;
    val = 34;
```

If I try to change the value of variable named 'val' it occurs a compiler error saying cannot assign twice to an immutable variable is occured

4 - In a constant ->

- const keyword is used when declaring a variable
- constants are immutable by default and cant use mut keyword with a constant
- constants are not allowed to change
- constants can decalre in any scope even in global scope
- constants are evaluated at compile time and cant assign a value at run time
- constants can accessible from anywhere within the scope which it declared

differences between immutable variables and constants ->

- constants are declared with const keyword while immutable variables are declared with let keyword
- constant with same name cant define multiple times but immutable variables with same name can define multiple times

```
const TOTAL_STUDENTS:i32 = 400;
    // cant define multiple times
    const TOTAL_STUDENTS:i32 = 500;

    println!("{TOTAL_STUDENTS}");

    let value = 10;
    // can define multiple times
    let value = 12;
    println!("{value}");
```

5 -

```
let val = 10;

    let val = val + 4;

    println!("{val}"); output is 14
```

like in the above example if I declare two variables with the same name, the first variable is known as it is shadowed by the second variable. It means that compiler only see the value of second variable and it is over shadow the first variable.

6 –

```
let val = 10;

    let val = "Hello";

    println!("{val}"); output is Hello
```

Yes, as the above example we can change the variable’s type during shadowing.

7 –

```
let val:&str;
    println!("{val}");
```

can’t use uninitialized variables in rust. As above example compile time error occurred.

8 –

By only using let keyword variable is declared as an immutable variable which cannot change the value.

But if use let mut keyword variable can declare a mutable variable which can change the value.

9 – In rust entire instance must be mutable. Rust doesn’t allow us to mark only certain fields as mutable.

```
fn main() {
    struct User {
        name: String,
        email: String,
    }

    // mutable instance
    let mut user = User {
        name: String::from(""),
        email: String::from("risni.jeewa@gmail.com"),
    };
    // can assign new values for user instance of User
    user.name = String::from("Risni");

    println!("name:{}, email: {}", user.name, user.email);

    // immutable instance
    let user2 = User{
        name:String::from("Risni"),
        email: String::from("risni.jeewa@gmail.com"),
    };

    // cant assign to user2 instance as its an immutable instance of User
    user2.name = String::from("Ruvi");
}
```

10 – Only variables with const and static keywords can declare outside a fuction . But variable with let or let mut cant declare outside a function.

```
static val:i32 = 10;
const val2:i32 = 100;

fn main() {
    println!("{val}"); output is 10

}
```

11-

- & is known as a reference and it allow us to refer to some value without taking ownership of it. So it allowed us to access to a data without transferring ownership.

```
let val = 10;
    let val2 = &val;

    println!("{val},{val2}"); output is 10, 10
```

So by referring to first value (val), val2 can modify that 'val' value.

- ref is used in pattern bindings to borrow a value rather than moving that value.

- So after the match statement also we can use value variable

```
•   let value = Some(42);
•       match value {
•           Some(ref v) => println!("Borrowed value: {}", v),
•           None => println!("No value"),
•       };
```

12 – Ownership is a set of rules that govern how a Rust program manages memory. Each value has a single owner at a time where when the owner goes out of scope then value will be dropped.

```
let val = String::from("hello");
    let val2 = val;

    println!("{},{}",val,val2);
```

in the above example error occurred due to val is no longer available because its ownership transferred to val2

13 -
Data which has a fixed size stored on the stack. Data with an unknown size at compile time or a size that might change stored on the heap.

Allocation and deallocation of data is faster in stack variables than heap variables.

Examples for stack variables – integers,floats

Examples for stack variables – Strings

14 – Closures aren't used in an exposed interface like functions. They're stored in variables and used without naming them and exposing them to users of our library. Closures can capture values from their environment in three ways.

1st way is closure reads the variable without modifying it which is an immutable borrow. In this variable can use outside the closure.

```
let x = 10;
    let closure = || {
        println!("x is {}", x); // Closure borrows `x` immutably
    };

    println!("x is still usable: {}", x);

    closure();
    println!("x is still usable: {}", x);
```

we can create multiple immutable references at the same time. therefore variable accessible from the code, before the closure and after the closure.

2nd way is closure borrows the variable mutably and allow to modify the value. Until closure exists this prevents other mutable or immutable accesses to the variable.

```rust
let mut x = 10;

    let mut closure = || {
        x += 1; // Closure mutably borrows `x`
        println!("x inside closure: {}", x);
    };

    let val = x;  // error: because x is mutably borrowed
    closure();
    println!("{}", x); // can access x now because closure is closed
```

As above example between closure definition and the closure call immutable borrow isn't allowed because no other borrows are allowed when there's a mutable borrow.

3rd way is closure taking ownership of the variable so it consumes the variable and cant use outside the closure even closure closed. move keyword is used to forcefully give the ownership of a variable in to a closure.

```rust
let x = String::from("Hello");

    let closure = move || {
        println!("x is owned inside the closure: {}", x); // Closure takes
ownership
    };

    closure();
    println!("{}", x);  // error: x is moved
```

a variable cant be mutably borrowed both inside and outside a closure at the same time. As below.

```rust
let mut x = 10;

    let mut closure = || x += 1; // Mutable borrow occurs here
    x += 1; // Error: `x` is already mutably borrowed by the closure

    closure();
```

Once a closure has captured a reference or captured ownership of a value from the environment where the closure is defined, the code in the body of the closure defines what happens to the references or values when the closure is evaluated.

A closure body can do move a captured value out of the closure or mutate the captured value or neither move nor mutate the value, or capture nothing from the environment to begin with. Depending on how the closure's body handles the values closures will automatically implement one, two, or all three of these Fn traits.

Fn is use when the closure only reads variables. These closures can be called more than once without mutating their environment. FnMut use when closure modifies variables. These closures can be called more than once. FnOnce use for closures that need to consume variables. This trait can only be called once.

```
fn call_closure<F: Fn()>(f: F) {

    f(); // Calls a closure that borrows immutably
}

fn call_closure_mut<F: FnMut()>(mut f: F) {
    f(); // Calls a closure that borrows mutably
}

fn call_closure_once<F: FnOnce()>(f: F) {
    f(); // Calls a closure that takes ownership
}

fn main() {

    let x = String::from("Immutable");
    call_closure(|| println!("{}", x)); // prints the immutable value of x which
is Immutable

    let mut y = 5;
    call_closure_mut(|| println!("{}", y)); // second line prints the mutable
value of y which is 5

    let z = String::from("Owned");
    call_closure_once(|| println!("{}", z)); // third line prints the owned value
of z which is owned.
    // After this, z cannot be accessed

}
```

15- Aliases refer to the situations where multiple references point to the same memory location which allows variables to be accessed through multiple paths or names. Aliased create using mutable or immutable references.

Immutable aliase ->

```
let val = 10;

    let val1 = &val;
    let val2 = &val;
    println!("{val1},{val2}");
```

multiple immutable references(val1,val2) to val variable are allowed at the same time.

Mutable aliase ->

```
let val = 10;

    let val1 = &mut val;
    let val2 = &mut val;
    println!("{val1},{val2}");
```

multiple mutable references to val variable is not allowed at the same time. But only one mutable reference can create and it allows us to modify the value of the val. And in that situation val variable is no longer valid until the mutable reference scope ends. If try to access val before mutable reference ends display compile error displaying its cannot borrow as immutable because mutably borrowed. Both immutable and mutable aliases cant exist at the same time.

```
let mut val = 10;

    let val1 = &mut val;
    *val1 += 1;
    println!("{},{}",val1,val); // cant access val error occur
    println!("{}",val); // can access val
```

In Rust, when you assign or pass a value, its **ownership** is transferred (moved) unless it's a **copyable type**. When copyable types like integers,floats, char assigned or passed, they are **copied** rather than moved. This means the original variable is still valid after the assignment.

```rust
let val = 10;

    let val1 = &val; // Immutable reference
    let val2 = val; // Copies the value

    println!("{}", val1);
    println!("{}", val2);
```

```rust
let data = String::from("Hello");

    let alias = &data; // Immutable reference
    let moved_data = data; // Error: Ownership of `data` would move here

    println!("{}", alias);
```

Aliases must be valid as long as the data they reference exists. Unsafe aliasing bypasses Rust's safety guarantees but must be used with caution.

16 – can define an infinite loop by using loop keyword. The scope inside the loop will run until code uses break or the program exits.

```rust
fn main() {
    loop {
        println!("hello");
    }
}
```

17 – By using break keyword inside the loop

```rust
fn main() {

    loop {
        println!("hello");
        break;
    }
}
```

18 - By using continue keyword inside the loop

```rust
fn main() {
    let mut i = 1;
    loop {
        println!("hello");
        i += 1;
        if (i == 3){
            continue;
        };
        if (i == 5){
            break;
        };

    }
}
```

In the above code when i == 3 it skips to next iteration

19 – Defining one loop inside another loop can create a nested loop in rust

```
fn main() {
    // let mut i = 1;
    for i in (1..3) {
        for j in (1..2){
            println!("outer :{}, inner:{}",i,j);
        }


    }
}
```

```
Output :        outer :1, inner:1

                outer :2, inner:1
```

20 –

```
fn main() {
    'outer: for i in 0..3 {
        for j in 0..3 {
            if i == 1 && j == 1 {
                break 'outer; // Break out of the outer loop
            }
            println!("i: {}, j: {}", i, j);
        }
    }

}
```

Output:

```
i: 0, j: 0

i: 0, j: 1

i: 0, j: 2

i: 1, j: 0
```

if remove the break statement

```
i: 0, j: 0

i: 0, j: 1

i: 0, j: 2

i: 1, j: 0

i: 1, j: 1

i: 1, j: 2

i: 2, j: 0

i: 2, j: 1

i: 2, j: 2
```

21 – Depending on how we want to interact with the elements of the vector there are several ways to iterate over a vector using for loop.

1st method – by value

in this method vector is directly use in the for loop therefore ownership of each element is transferred to the loop, and the vector is consumed. In this method after for loop we can't access the vector.

```rust
let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers {
        println!("{}", num);
    }
println!("{:?}", numbers); // error : already consumed
```

2nd method – by mutable reference

In this method borrows elements mutably and allows us to modify the elements of the vector in place. In this method after for loop we can access the vector.

```
let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers.iter_mut() {
        *num += 1;
        println!("{}", num);
    }
println!("{:?}", numbers);
```

3rd method – by immutable reference

In this method borrows elements immutably and don't allows us to modify the elements of the vector in place. In this method after for loop we can access the vector.

```
Let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers.iter() { //  cant do *num += 1 like modifications
        println!("{}", num); // Read-only access
    }
```

4th method -by Index

Use a for loop with the range syntax and access elements by their indices. This allows more control but requires indexing into the vector, which can be slower than iterators due to bounds checking. Vector remains usable after for loop .

```
let numbers = vec![1, 2, 3, 4, 5];
    let length = numbers.len();

    for i in 0..length {
        println!("{}", numbers[i]); // Access elements using their index
    }

    // The vector remains usable
    println!("{:?}", numbers);
```

5th method -By using enumerate

Combine .iter() with .enumerate() to get both the index and the value during iteration. This is useful when we need the position of elements.

```
let numbers = vec![1, 2, 3, 4, 5];
```

```
    for (index,num) in numbers.iter().enumerate() {
        println!("index: {}, number: {}", index,num);
    }

    // The vector remains usable
    println!("{:?}", numbers);
```

6th method -by using into_iter()

Use .into_iter() to consume the vector and take ownership of its elements. Similar to iterating by value, but often used when explicitly chaining iterator methods.

```
 let numbers = vec![1, 2, 3, 4, 5];

    for num in numbers.into_iter() {
        println!("{}", num); // Takes ownership of each element
    }

    // Error: `numbers` is moved and cannot be used here
    println!("{:?}", numbers);
```

22 – Iterating by reference and by value in a for loop differ in how ownership and borrowing are handled for the elements of the collection. In iterationg by value method ownership is passed to each value in the collection, so after the for loop cant access that collection ‘numbers’.

```
let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers {
        println!("{}", num);
    }
    println!("{:?}", numbers); // `numbers` is moved and cannot be used here
```

In iterating by reference method only reference to each element is given so after the for loop we can access the collection ‘numbers’. There are two methods to do iterating by reference which are iter() and iter_mut(). In iter() only read access given during iteration while in iter_mut() provides read and write access.

```
let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers.iter() {
        println!("{}", num);
    }
    println!("{:?}", numbers);

    for num in numbers.iter_mut() {
        *num += 1;
        println!("{}", num);
    }
    println!("{:?}", numbers);
```

23 – iter() and iter_mut() methods used to create **iterators** for traversing elements in arrays, vectors etc.To create an immutable iterator iter() method used. So during the iteration it only allows us to read cant modify.

```
Let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers.iter() { //  cant do *num += 1 like modifications
        println!("{}", num); // Read-only access
    }
```

iter_mut() is a mutable iterator which allows us to modify the elements during the iteration.

```
let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers.iter_mut() {
        *num += 1;
        println!("{}", num);
    }
```

24 – To optimize loops in rust we can follow several methods. First way is we can compile code with –release mode for faster performance. Cargo has two main profiles:
the dev profile Cargo uses when we run cargo build and the release profile Cargo uses when we run cargo build –release

```
cargo build --release
```

another way is by using iterators, which eliminate manual indexing and bounds checking. So loops will faster and safer. Also iterators avoid runtime bounds checks compared to manual indexing.

```
let mut numbers = vec![1, 2, 3, 4, 5];
    for num in numbers.iter () {

        println!("{}", num);
    }
```

Another way is using enumerate for indexes. When we need both index and value of a vector or array this can use to avoid manual indexing.

```
let numbers = vec![1, 2, 3, 4, 5];
```

```
    for (index,num) in numbers.iter().enumerate() {
        println!("index: {}, number: {}", index,num);
    }
```

Another instance when working with complex types during an iteration borrow data instead of copying.

```
let vec = vec![String::from("Hello"), String::from("World")];
    for value in vec {
        println!("{}", value);
    }
```

Another instance is when we don't need loop control or indexes to optimize loops can use .for_each() method

```
let vec = vec![1, 2, 3, 4, 5];
    vec.iter().for_each(|x| println!("{}", x));
```

To optimize loops we need to avoid resizing during iteration, for that we can pre-allocate memory for collections like below example.

```
let mut vec = Vec::with_capacity(100);
    for i in 0..100 {
        vec.push(i);
    }
    println!("{:?}",vec);
```

25 – There are some reasons where for loop and while loop differ in performance.
1st reason – use of iterators
In for loops which are optimized at compile time. Therefore it eliminates unnecessary runtime checks and using for loops can traverse a collection faster. But while loops rely on manual conditions. So may involve runtime checks like bounds checking for indexed access.

```
let vector = vec![1,2,3,4,5];

    for i in vector.iter(){  // loop operate on iterators
        print!("{} ",i);
    }
```

```rust
let vector = vec![10,20,30,40];

    let mut i = 0;
    while i < vector.len(){   // loop run until condition true
        print!("{} -> {} ",i,vector[i]);
        i += 1;
    }
```

2nd reason – bounds checking

in for loops it doesn't occur additional overhead because it ensure safety and prevent out of bounds access. But in while loops like in the above example bounds must be checked explicitly.

3rd reason - Compiler Optimization.

In for loops compiler optimizes more effectively due to predictable nature of iterators. But while loops are less optimized because of its dynamic conditions and flexibility.

4th reason - Error-Prone Nature

In for loopsinternally handle iteration and minimize errors. But in while loops errors occur if indexing or conditions not handle efficiently.

26 - true

27 –

```rust
fn main() {

    let mut x = 10;
    x += 5;

    println!("x : {}",x)

}
```

28 -

```rust
fn main() {
    println!("Enter a number");
    let mut num = String::new();

    io::stdin()
    .read_line(&mut num)
    .expect("Failed to read the input");

    let x: i32 = num
        .trim()
        .parse()
        .expect("Please enter a valid number");

    let y = 3 * x * x + 2 * x + 1;
    println!("y: {}",y);

}
```