Rust - Day 3

Variables and Mutability

- **Variables are immutable by default in rust.** It helps write code in a way that takes advantage of the safety and easy concurrency that Rust offers.
- Rust **encourages you to favor immutability** but sometimes you might want to opt out.

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

- This would throw **immutability error** because the **same variable x (immutable by default) is being reassigned.**
- This behavior helps write bug free code. For example
  - A certain part of the code excepts a value of a variable to never change while another part changes it and worse of it changes it sometimes (maybe based off some calcuation or api response). This can lead to unexpected behaviors which would be hard to comprehend.
- To make variables mutable use `mut` keyword in front of them.

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

- This code would run fine as the variable is marked as mutable.
- **Deciding whether to use mutability or not is up to you and depends on what you think is clearest in that particular situation.**

Constants

- Constants are values that are bound to a name and are not allowed to change.
- **Constants are always immutable therefore `mut` cannot be used with them.**
- Declared using `const` instead of `let` keyword
- **Type of constant value must be defined during creation**
- Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of code need to know about.\
- **The last difference is that constants may be set only to a constant expression, not the result of a value that could only be computed at runtime.**
- Rust's naming convention for constants is to use all uppercase with underscores between words like:

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- Rust compiler can evaluate a limited set of operations at compile time which helps us define these expressions as used above instead of using 10,800 value.
- **Constants are valid for the entire time a program runs, within the scope in which they were declared.**

Shadowing

- When you reassign the value of a variable (immutable) to another with the same name then the previous value gets shadowed and would continue until either this new instance is shadowed or the scope ends.

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

- Shadowing is different from marking a variable as `mut` because we'll get a compile-time error if we accidentally try to reassign to this variable without using the `let` keyword. **By using `let`, we can perform a few transformations on a value but have the variable be immutable after those transformations have been completed.**
- **The other difference between `mut` and shadowing is that because we're effectively creating a new variable when we use the `let` keyword again, we can change the type of the value but reuse the same name.**

```
    let spaces = "   ";
    let spaces = spaces.len();
```

- The first `spaces` variable is a string type and the second `spaces` variable is a number type.
- **Shadowing thus spares us from having to come up with different names, such as `spaces_str` and `spaces_num`**
- While this would throw error

```
    let mut spaces = "   ";
    spaces = spaces.len();
```

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
2 |     let mut spaces = "   ";
  |                      ----- expected due to this value
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

Data Types

- **Rust is a *statically typed* language**, which means that it must know the types of all variables at compile time.
- The compiler can usually **infer** what **type** we want to use based on the value and how we use it.
- This is how you add type annotation

```
let guess: u32 = "42".parse().expect("Not a number!");
```

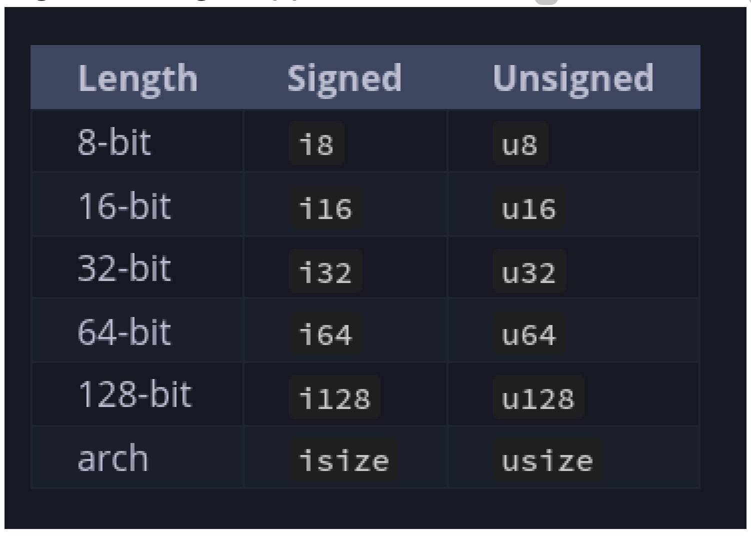
- If we don't define the type while parsing the value error would be thrown as compiler won't know which datatype to parse to.

Scalar Types

- A *scalar* type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters.

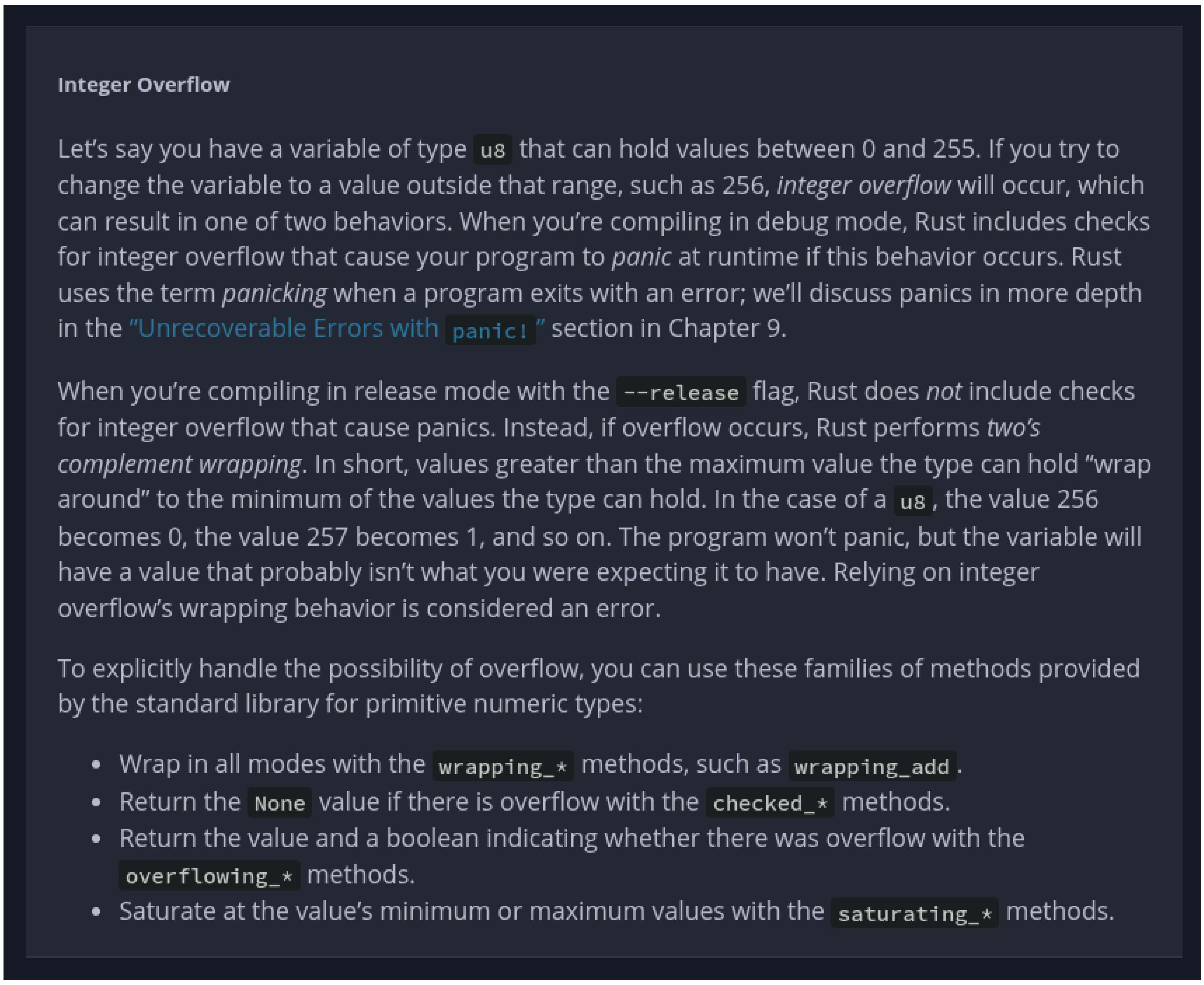
Integer Types

- An *integer* is a number without a fractional component.
- signed integer types start with `i` instead of `u`



Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

- *Signed* and *unsigned* refer to whether it's possible for the number to be negative—in other words, whether the number needs to have a sign with it (signed) or whether it will only ever be positive and can therefore be represented without a sign (unsigned).
    - **Signed** - It can be positive or negative
    - **Unsigned** - Always positive
- Signed numbers are stored using **Two's compliment** representation
- **Each signed variant can store numbers from -(2n - 1) to 2n - 1 - 1 inclusive, where *n* is the number of bits that variant uses. So an `i8` can store numbers from -(27) to 27 - 1, which equals -128 to 127. Unsigned variants can store numbers from 0 to 2n - 1, so a `u8` can store numbers from 0 to 28 - 1, which equals 0 to 255.**
- Additionally, the `isize` and `usize` types depend on the architecture of the computer your program is running on, which is denoted in the table as "arch": 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.
- **literals can also use `_` as a visual separator to make the number easier to read, such as `1_000`, which will have the same value as if you had specified `1000`.**
- **integer types default to `i32`.**



Integer Overflow

Let's say you have a variable of type `u8` that can hold values between 0 and 255. If you try to change the variable to a value outside that range, such as 256, *integer overflow* will occur, which can result in one of two behaviors. When you're compiling in debug mode, Rust includes checks for integer overflow that cause your program to *panic* at runtime if this behavior occurs. Rust uses the term *panicking* when a program exits with an error; we'll discuss panics in more depth in the "Unrecoverable Errors with `panic!`" section in Chapter 9.

When you're compiling in release mode with the `--release` flag, Rust does *not* include checks for integer overflow that cause panics. Instead, if overflow occurs, Rust performs *two's complement wrapping*. In short, values greater than the maximum value the type can hold "wrap around" to the minimum of the values the type can hold. In the case of a `u8`, the value 256 becomes 0, the value 257 becomes 1, and so on. The program won't panic, but the variable will have a value that probably isn't what you were expecting it to have. Relying on integer overflow's wrapping behavior is considered an error.

To explicitly handle the possibility of overflow, you can use these families of methods provided by the standard library for primitive numeric types:

- Wrap in all modes with the `wrapping_*` methods, such as `wrapping_add`.
- Return the `None` value if there is overflow with the `checked_*` methods.
- Return the value and a boolean indicating whether there was overflow with the `overflowing_*` methods.
- Saturate at the value's minimum or maximum values with the `saturating_*` methods.

Floating-Point Types

- Rust also has two primitive types for *floating-point numbers*
- Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively.
- default type is `f64`
- Floating-point numbers are represented according to the IEEE-754 standard. The `f32` type is a single-precision float, and `f64` has double precision.

Numeric Operations

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

Boolean Type

- **1 byte in size**
- Specified using `bool`

```
fn main() {
    let t = true;
    let f: bool = false; // with explicit type annotation
}
```

Character Type

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

- Specified using **single quotes** unlike strings which use double quotes
- **4 bytes in size and represents a Unicode Scalar Value** which means it can represent a lot more than just ASCII
    - Accented letters; Chinese, Japanese, and Korean characters; emoji; and zero-width spaces are all valid `char` values in Rust.

Compound Types

- Rust has two primitive compound types: **tuples** and **arrays**.

Tuple Type

- A *tuple* is a general way of grouping together a number of values with a variety of types into one compound type.
- **Tuples have a fixed length**
- Once declared, they cannot grow or shrink in size.
- We create a tuple by writing a comma-separated list of values inside parentheses. **Each position in the tuple has a type, and the types of the different values in the tuple don't have to be the same.**

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

- The type annotations are optional here.
- **To get the individual values out of a tuple, we can use pattern matching to destructure a tuple value**

```
fn main() {
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of y is: {y}");
}
```

- This program first creates a tuple and binds it to the variable `tup`. It then uses a pattern with `let` to take `tup` and turn it into three separate variables, `x`, `y`, and `z`. **This is called *destructuring* because it breaks the single tuple into three parts.**
- **We can also access a tuple element directly by using a period (`.`) followed by the index of the value we want to access.**

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);
    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;
}
```

- The tuple without any values has a special name, *unit*. This value and its corresponding type are both written `()` and represent an empty value or an empty return type. **Expressions implicitly return the unit value if they don't return any other value.**

Array Type

- Every element of an array must have the same type
- They also have a fixed length

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

- **Arrays are useful when you want your data allocated on the stack, the same as the other types we have seen so far, rather than the heap or when you want to ensure you always have a fixed number of elements. An array isn't as flexible as the vector type, though.**
- **A *vector* is a similar collection type provided by the standard library that *is* allowed to grow or shrink in size.** If you're unsure whether to use an array or a vector, chances are you should use a vector.
- **Array - Fixed Size, Same Datatypes**
- **Vector - Variable Size, Same Datatypes**
- You write an array's type using square brackets with the type of each element, a semicolon, and then the number of elements in the array, like so:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

- You can also initialize an array to contain the same value for each element by specifying the initial value, followed by a semicolon, and then the length of the array in square brackets, as shown here:

```
let a = [3; 5];
```

- The array named `a` will contain `5` elements that will all be set to the value `3` initially. This is the same as writing `let a = [3, 3, 3, 3, 3];` but in a more concise way.
- Accessing array values

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

- **In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed. Rust protects you against this kind of error by immediately exiting instead of allowing the memory access and continuing. Chapter 9 discusses more of Rust's error handling and how you can write readable, safe code that neither panics nor allows invalid memory access.**

Functions

- The `main` function is the entry point
- Rust code uses *snake case* as the conventional style for function and variable names

```
fn main() {
    println!("Hello, world!");

    another_function();
}
```

```
fn another_function() {
    println!("Another function.");
}
```

Parameters

- Type of params must be defined

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

- Passing multiple params

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Statements and Expressions

- Function bodies are made up of a series of statements optionally ending in an expression.
- Rust is an expression-based language.
- **Statements** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value.
- **Creating a variable and assigning a value to it with the `let` keyword is a statement. In Listing 3-1, `let y = 6;` is a statement.**
- Statements do not return values. Therefore, you can't assign a `let` statement to another variable, as the following code tries to do; you'll get an error:

```
fn main() {
    let x = (let y = 6);
}
```

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: only supported directly in conditions of `if` and `while` expressions

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -    let x = (let y = 6);
2 +    let x = let y = 6;
  |

warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` (bin "functions") due to 1 previous error; 1 warning emitted
```

- The `let y = 6` statement does not return a value, so there isn't anything for `x` to bind to. This is different from what happens in other languages, such as C and Ruby, where the assignment returns the value of the assignment. **In those languages, you can write `x = y = 6` and have both `x` and `y` have the value `6`; that is not the case in Rust.**
- Expressions evaluate to a value and make up most of the rest of the code that you'll write in Rust. Consider a math operation, such as `5 + 6`, which is an expression that evaluates to the value `11`. **Expressions can be part of statements: in Listing 3-1, the `6` in the statement `let y = 6;` is an expression that evaluates to the value `6`. Calling a function is an expression. Calling a macro is an expression. A new scope block created with curly brackets is an expression**, for example:

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

- This expression below evaluates to 4 and gets bound to y as part of `let` statement

```
{
    let x = 3;
    x + 1
}
```

- **Note that the `x + 1` line doesn't have a semicolon at the end, which is unlike most of the lines you've seen so far. Expressions do not include ending semicolons. If you add a semicolon to the end of an expression, you turn it into a statement, and it will then not return a value.**

Functions with return values

```
fn five() → i32 {
    5
}

fn main() {
    let x = five();
    println!("The value of x is: {x}");
}
```

- There are no function calls, macros, or even `let` statements in the `five` function—just the number `5` by itself. That's a perfectly valid function in Rust. Note that the function's return type is specified too, as `→ i32`.
- `5` with no semicolon because it's an expression whose value we want to return.

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) → i32 {
    x + 1
}
```

- Running this code will print `The value of x is: 6`. But if we place a semicolon at the end of the line containing `x + 1`, changing it from an expression to a statement, we'll get an error:

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` (bin "functions") due to 1 previous error
```

- **Better to use `return` keyword**

Comments

- Single line comment

```
// hello, world
```

- Multi line comment

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
```

- **Rust also has another kind of comment, documentation comments**

Control Flow

- Conditional behaviors and Looping

if Expressions

- **Conditions must be of type `bool` unlike JS which also makes nullish checks.**

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Using if in a let statement

- Because `if` is an expression, we can use it on the right side of a `let` statement to assign the outcome to a variable

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

```
The value of number is: 5
```

- Results from each arm of the `if` must be the same type unlike here:

```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "six" };
  |                                 -          ^^^^^ expected integer, found `&str`
  |                                 |
  |                                 expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

Looping

- Rust has three kinds of loops: `loop`, `while`, and `for`.

loop

- The `loop` keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.

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

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

- You can place the `break` keyword within the loop to tell the program when to stop executing the loop.

Returning values from loops

- One of the usecase of `loop` is to retry an operation until it succeeds and be able to pass the result of the loop to rest of your code. To do this, you can add the value you want returned after the `break` expression you use to stop the loop; that value will be returned out of the loop so you can use it, as shown here:

```
fn main() {
    let mut counter = 0;
    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;
        }
    };
    println!("The result is {result}");
}
```

- Before the loop, we declare a variable named `counter` and initialize it to `0`. Then we declare a variable named `result` to hold the value returned from the loop. On every iteration of the loop, we add `1` to the `counter` variable, and then check whether the `counter` is equal to `10`. When it is, we use the `break` keyword with the value `counter * 2`. After the loop, we use a semicolon to end the statement that assigns the value to `result`. Finally, we print the value in `result`, which in this case is `20`.
- **While `break` only exits the current loop, `return` always exits the current function.**

Loop Labels

- If you have loops within loops, `break` and `continue` apply to the innermost loop at that point. You can optionally specify a *loop label* on a loop that you can then use with `break` or `continue` to specify that those keywords apply to the labeled loop instead of the innermost loop. **Loop labels must begin with a single quote.**

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;
        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Conditional loops with while

```
fn main() {
    let mut number = 3;
    while number != 0 {
        println!("{number}!");

        number -= 1;
    }
    println!("LIFTOFF!!!");
}
```

Looping through a collection with for

- You can also use the `while` construct to loop over the elements of a collection, such as an array.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);
        index += 1;
    }
}
```

- You can use a `for` loop and execute some code for each item in a collection.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for element in a {
        println!("the value is: {element}");
    }
}
```

- Here's what the countdown would look like using a `for` loop and another method we've not yet talked about, `rev`, to reverse the range:

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```