```rust
/// The `main` function will be executed first in a binary.
/// Triple-slash comments are documentation -- check out `cargo doc`
fn main() {
    getting_started();          // basic syntax, simple types
    custom_types();             // define custom types and add methods to them.
    ownership_and_borrowing(); // *the* feature of Rust
}

fn getting_started() {
    let x = 1;                  // `x` is now `1` (a 32bit integer, `i32`)
    let mut y = 2;              // A mutable binding: We can change this value
    y += 1;                     // Add 1 to `y`, making it 3

    let x = "Hello";            // A reference to a string slice (`&str`)
    let mut y = x.to_owned(); // Copy the text into a mutable `String`.
    y.push_str(" World!");      // Now, we can change it!

    let xs = vec![7, 3, 8];     // A list (vector) of integers: `Vec<i32>`
    for x in &xs {              // Iterate over the list of values (by reference)
        println!("{}!", x);     // Print text to `stdout`, replacing `{}` with…
    }                           // …the value of `x` (using the `Display` trait)
}

fn custom_types() {
    struct Person {             // Define a new data type with two fields,…
        name: String,           // `name`, which contains a `String`, and
        age: u16,               // `age`, which is a unsigned 16bit integer
    }

    let jim = Person { name: "Jim".to_owned(), age: 25 };

    impl Person {               // Add inherent methods to the type
        fn new(name: &str, age: u16) -> Person {
            let name = if name.is_empty() { // `if` is an expression.
                "anonymous".to_owned()     // Both branches need to return
            } else {                       // the same type
                name.to_owned()
            };                             // Oh, and names can be shadowed

            Person { name, age } // Shortcut syntax!
        } // The last expression, the `Person` definition here, is returned…
    }     // …by the block. You can also write `return` (e.g. in early returns)

    use std::fmt;      // Import the formatting module from the standard library
    impl fmt::Display for Person { // Implement this trait for our new type
        /// Introduce the person
        fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
            write!(f, "Hi, I'm {}. I'm {} years old", self.name, self.age)
        }
    }

    println!("{}", jim);
}

fn ownership_and_borrowing() {
    fn this_borrows(x: &str) -> bool {
        x.len() > 5
    }
    let name = "Gustav".to_owned();
    this_borrows(&name);            // Pass a reference to the function
    println!("{}", name);           // `name` is still available, and unchanged

    let mut name = "Johan".to_owned();
    fn this_borrows_mutable(x: &mut String) {
        x.push_str(", nice to meet you!");
    }
    this_borrows_mutable(&mut name);
    println!("{}", name);           // "Johan, nice to meet you!"

    fn this_consumes(x: String) -> String {
        "I ate your string!".to_owned()
    }
    let name = "Peter".to_owned();
    let data = this_consumes(name); // After this, `name` will be unavailable!
    //> this_borrows(&name);        // Error: "borrow of moved value: `name`"
    println!("{}", data);           // The compiler errors are nice. Try it!
}
```