```
///////////////
// 1. Basics //
///////////////

// Functions. ‘i32’ is the type for 32-bit signed integers
fn add2(x: i32, y: i32) -> i32 {
    // Implicit return (no semicolon)
    x + y
    // Can also use explicit return: return x + y;
    // Call this function: add2(1, 3)
}

// Main function
fn main() {
    // Numbers //

    // Immutable bindings
    let x: i32 = 1;
    // x = 3; <-- compile-time error

    // Mutable variable
    let mut mutable = 1;
    mutable = 4;
    mutable += 2;

    // Integer/float suffixes
    let y: i32 = 13i32;
    let f: f64 = 1.3f64;

    // Type inference
    //
    // Most of the time, the Rust compiler can infer what type a variable is, so
    // you don’t have to write an explicit type annotation. Throughout this
    // tutorial, types are explicitly annotated in many places for demonstrative
    // purposes. Type inference can handle this for you most of the time.
    let implicit_x = 1;
    let implicit_f = 1.3;

    // Arithmetic
    let sum = x + y + 13;

    // Strings //

    // String literals
    let x: &str = "hello world!";

    // Printing
    println!("{} {}", f, x); // 1.3 hello world

    // A ‘String’ - a heap-allocated string
    let s: String = "hello world".into();
    let s2: String = "hello world".to_string();
    let s3: String = String::from("hello world");

    // A string slice: an immutable view into another string.
    //
    // This is essentially an immutable pair of pointers to a string - it
    // doesn’t actually contain the contents of a string, just a pointer to the
    // begin and a pointer to the end of a string buffer, statically allocated
    // or contained in another object (in this case, ‘s’)
    let s_slice: &str = &s;
    let s_slice2: &str = &s[6..11];
    let s_slice3: &str = &s[6..];
    let s_slice4: &str = &s[..5];

    println!("{} {}", s, s_slice); // hello world hello world

    // Vectors/arrays //

    // A fixed-size array
    let four_ints: [i32; 4] = [1, 2, 3, 4];

    // A dynamic array (vector)
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    vector.push(5);

    // Mutability is inherited by the bound value. If ‘vector‘ is not declared
    // ‘mut‘, then the value cannot be mutated.
    let vector: Vec<i32> = vec![1, 2, 3, 4, 5];
    // vector.push(5); <-- compile-time error

    // A slice - an immutable view into a vector or array.
    let slice: &[i32] = &vector;
    let slice2: &[i32] = &vector[1..4];

    // Use ‘{:?}‘ to print something debug-style
    println!("{:?} | {:?}", vector, slice2); // [1, 2, 3, 4, 5] | [2, 3, 4]

    // Array, slice, and vector indexing.
    println!("{}", four_ints[1]); // 2
    println!("{}", vector[2]); // 3
    println!("{}", slice[3]); // 4

    // Tuples //

    // A tuple is a fixed-size set of values of possibly different types
    let x: (i32, &str, f64) = (1, "hello", 3.4);

    // Destructuring ‘let‘
    let (a, b, c) = x;
    println!("{} {} {}", a, b, c); // 1 hello 3.4
    // Structures can also be destructured on assignment, as we’ll see later.

    // Tuple indexing.
    println!("{}", x.1); // hello

    //////////////
    // 2. Types //
    //////////////

    // Struct
    struct Point3 {
        x: i32,
        y: i32,
        z: i32,
    }

    let origin: Point3 = Point3 { x: 0, y: 0, z: 0 };

    // A struct with unnamed fields, called a "tuple struct"
    struct Point2(i32, i32);

    let origin2 = Point2(0, 0);

    // Basic C-like enum
    enum Direction {
        Left,
        Right,
        Up,
        Down,
    }

    let up = Direction::Up;
    let down = Direction::Down;

    // Enum with fields. Variants can be nullary, tuple structs, or structs.
    enum Message {
        Quit,
        Write(String),
        Move { x: i32, y: i32 },
    }

    let quit: Message = Message::Quit;
    let write: Message = Message::Write("Hello!".into());
    let mov: Message = Message::Move { x: 20, y: 120 };

    /////////////////////////
    // 3. Pattern matching //
    /////////////////////////

    match mov {
        Message::Quit => println!("quitting..."),
        Message::Write(s) => println!("Writing: {}", s),
        Message::Move { x, y } => println!("Move to: ({}, {})", x, y),
    }

    // Advanced pattern matching
    struct FooBar { x: i32, y: Message }
    let bar = FooBar { x: 15, y: Message::Quit };

    match bar {
        FooBar { x: 0, y: Message::Quit } => println!("Quitting with x = 0!"),
        FooBar { x: 2, .. } => println!("x is 2"),
        FooBar { x: x1, y: Message::Move { x: x2, y } } if x1 == x2 => {
            println!("x’s match! y = {}", y);
        }
        _ => println!("sink for everything unmatched"),
    }

    /////////////////
    // 4. Generics //
    /////////////////

    // A structure with a field of generic type ‘T’.
    struct Foo<T> { bar: T }

    // This is a type alias; not a new type, just another name for it.
    type FooI32 = Foo<i32>;
    let x: FooI32 = Foo { bar: 12 };
    let y: Foo<i32> = x;

    // This is defined in the standard library as ‘Option‘
    enum MyOption<T> {
        Some(T),
        None,
    }

    // This is defined in the standard library as ‘Result‘
    enum MyResult<T, E> {
        Ok(T),
        Err(E),
    }

    // Methods //

    impl<T> Foo<T> {
        // Static methods do not take a ‘self’ parameter.
        // let foo: Foo<i32> = Foo::new(123);
        fn new(bar: T) -> Foo<T> {
            Foo { bar: bar }
        }

        // Instance methods take an explicit ‘self’ parameter
        // let foo = Foo { bar: 123 };
        // let bar: i32 = foo.bar();
        fn bar(self) -> T {
            self.bar
        }
    }

    // Traits (known as interfaces or typeclasses in other languages) //
    trait Frobnicate<T> {
        fn frobnicate(self) -> Option<T>;
    }

    // Trait implementation.
    impl<T> Frobnicate<T> for Foo<T> {
        fn frobnicate(self) -> Option<T> {
            Some(self.bar)
        }
    }

    let another_foo = Foo { bar: 1 };
    println!("{:?}", another_foo.frobnicate()); // Some(1)

    // Traits can require implementors to implement other traits.
    trait Fabulous<T>: Frobnicate<T> {
        // ‘Self’ is a stand-in type for the type implementing this trait.
        fn fab(self) -> Self;
    }

    /////////////////////
    // 5. Control flow //
    /////////////////////

    // ‘for’ loops/iteration
    let array = [1, 2, 3];
    for i in array.iter() {
        println!("{}", i);
    }

    // Ranges: prints ‘0 1 2 3 4 5 6 7 8 9 ’
    for i in 0..10 {
        print!("{} ", i);
    }

    // ‘if’
    if 1 == 1 {
        println!("Math works!");
    } else {
        println!("Oh no...");
    }

    // ‘if’ as expression
    let value = if true {
        "good"
    } else {
        "bad"
    };

    // ‘while‘ loop
    let mut x = 0;
    while x < 10 {
        x += 1;
        if x == 5 {
            continue;
        }

        println!("x = {}", x);
    }

    // Infinite loop. Need to ‘break‘ explicitly.
    loop {
        println!("Hello!");
    }

    ///////////////////////////////////
    // 6. ‘Copy‘ and move semantics. //
    ///////////////////////////////////

    struct FooBoo(i32);

    // Can only have one binding to a value at a time. On new binding, value
    // gets "moved" and old binding is not useable.
    let x = FooBoo(1); // x "owns" FooBoo(1)
    let y = x; // y now owns FooBoo(1)
    // let z = x; <-- compile-time error (x moved to y)

    // Unless the type implements the ‘Copy‘ trait.
    // This trait is declared in the Rust core library.
    pub trait Copy: Clone { }

    // ‘derive‘ automatically generates implementations for traits
    #[derive(Copy, Clone)]
    struct Bar(i32);

    // Now value is copied instead of moved.
    let x = Bar(2);
    let y = x;
    let z = x;

    // All integer and float types are ‘Copy‘.
    let x = 1;
    let y = x;
    let z = x;

    // So are references.
    let a = &x;
    let b = a;
    let c = a;

    //////////////////////////////////////
    // 7. "Object-Oriented" Programming //
    //////////////////////////////////////

    #[derive(Debug)]
    struct Point {
        x: i32,
        y: i32
    }

    // C-style OOP
    fn point_add(a: Point, b: Point) -> Point {
        Point { x: a.x + b.x, y: a.y + b.y }
    }
```

```rust

    // Java-style OOP
    impl Point {
        pub fn new(x: i32, y: i32) -> Point {
            Point { x, y }
        }

        // ‘&self‘: an immutable reference to ‘self‘
        // Can read ‘self‘ but not write to it.
        pub fn add(&self, other: Point) -> Point {
            Point { x: self.x + other.x, y: self.y + other.y }
        }

        // ‘&mut self‘: a mutable reference to ‘self‘
        // Can read and write to ‘self‘.
        pub fn set_x(&mut self, x: i32) {
            self.x = x;
        }
    }

    // ‘mut‘ is needed to create an ‘&mut‘ reference
    let mut p1 = Point::new(5, 2);

    // the ‘&mut‘ reference is automatically created on method call
    p1.set_x(10);

    let p2 = Point::new(3, 1);
    println!("{:?}", p1.add(p2));
}
```