```rust
///////////////
// 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,
```

```
131:     }
132:
133:     let up = Direction::Up;
134:     let down = Direction::Down;
135:
136:     // Enum with fields. Variants can be nullary, tuple structs, or structs.
137:     enum Message {
138:         Quit,
139:         Write(String),
140:         Move { x: i32, y: i32 },
141:     }
142:
143:     let quit: Message = Message::Quit;
144:     let write: Message = Message::Write("Hello!".into());
145:     let mov: Message = Message::Move { x: 20, y: 120 };
146:
147:     ////////////////////////
148:     // 3. Pattern matching //
149:     ////////////////////////
150:
151:     match mov {
152:         Message::Quit => println!("quitting..."),
153:         Message::Write(s) => println!("Writing: {}", s),
154:         Message::Move { x, y } => println!("Move to: ({}, {})", x, y),
155:     }
156:
157:     // Advanced pattern matching
158:     struct FooBar { x: i32, y: Message }
159:     let bar = FooBar { x: 15, y: Message::Quit };
160:
161:     match bar {
162:         FooBar { x: 0, y: Message::Quit } => println!("Quitting with x = 0!"),
163:         FooBar { x: 2, .. } => println!("x is 2"),
164:         FooBar { x: x1, y: Message::Move { x: x2, y } } if x1 == x2 => {
165:             println!("x's match! y = {}", y);
166:         }
167:         _ => println!("sink for everything unmatched"),
168:     }
169:
170:     /////////////////
171:     // 4. Generics //
172:     /////////////////
173:
174:     // A structure with a field of generic type `T`.
175:     struct Foo<T> { bar: T }
176:
177:     // This is a type alias; not a new type, just another name for it.
178:     type FooI32 = Foo<i32>;
179:     let x: FooI32 = Foo { bar: 12 };
180:     let y: Foo<i32> = x;
181:
182:     // This is defined in the standard library as `Option`
183:     enum MyOption<T> {
184:         Some(T),
185:         None,
186:     }
187:
188:     // This is defined in the standard library as `Result`
189:     enum MyResult<T, E> {
190:         Ok(T),
191:         Err(E),
192:     }
193:
194:     // Methods //
195:
```

```
196:     impl<T> Foo<T> {
197:         // Static methods do not take a 'self' parameter.
198:         // let foo: Foo<i32> = Foo::new(123);
199:         fn new(bar: T) -> Foo<T> {
200:             Foo { bar: bar }
201:         }
202:
203:         // Instance methods take an explicit 'self' parameter
204:         // let foo = Foo { bar: 123 };
205:         // let bar: i32 = foo.bar();
206:         fn bar(self) -> T {
207:             self.bar
208:         }
209:     }
210:
211:     // Traits (known as interfaces or typeclasses in other languages) //
212:     trait Frobnicate<T> {
213:         fn frobnicate(self) -> Option<T>;
214:     }
215:
216:     // Trait implementation.
217:     impl<T> Frobnicate<T> for Foo<T> {
218:         fn frobnicate(self) -> Option<T> {
219:             Some(self.bar)
220:         }
221:     }
222:
223:     let another_foo = Foo { bar: 1 };
224:     println!("{:?}", another_foo.frobnicate()); // Some(1)
225:
226:     // Traits can require implementors to implement other traits.
227:     trait Fabulous<T>: Frobnicate<T> {
228:         // 'Self' is a stand-in type for the type implementing this trait.
229:         fn fab(self) -> Self;
230:     }
231:
232:     ////////////////////
233:     // 5. Control flow //
234:     ////////////////////
235:
236:     // 'for' loops/iteration
237:     let array = [1, 2, 3];
238:     for i in array.iter() {
239:         println!("{}", i);
240:     }
241:
242:     // Ranges: prints '0 1 2 3 4 5 6 7 8 9 '
243:     for i in 0..10 {
244:         print!("{} ", i);
245:     }
246:
247:     // 'if'
248:     if 1 == 1 {
249:         println!("Math works!");
250:     } else {
251:         println!("Oh no...");
252:     }
253:
254:     // 'if' as expression
255:     let value = if true {
256:         "good"
257:     } else {
258:         "bad"
259:     };
260:
```

```
261:     // ‘while‘ loop
262:     let mut x = 0;
263:     while x < 10 {
264:         x += 1;
265:         if x == 5 {
266:             continue;
267:         }
268:
269:         println!("x = {}", x);
270:     }
271:
272:     // Infinite loop. Need to ‘break‘ explicitly.
273:     loop {
274:         println!("Hello!");
275:     }
276:
277:     ////////////////////////////////
278:     // 6. ‘Copy‘ and move semantics. //
279:     ////////////////////////////////
280:
281:     struct FooBoo(i32);
282:
283:     // Can only have one binding to a value at a time. On new binding, value
284:     // gets "moved" and old binding is not useable.
285:     let x = FooBoo(1); // x "owns" FooBoo(1)
286:     let y = x; // y now owns FooBoo(1)
287:     // let z = x; <-- compile-time error (x moved to y)
288:
289:     // Unless the type implements the ‘Copy‘ trait.
290:     // This trait is declared in the Rust core library.
291:     pub trait Copy: Clone { }
292:
293:     // ‘derive‘ automatically generates implementations for traits
294:     #[derive(Copy, Clone)]
295:     struct Bar(i32);
296:
297:     // Now value is copied instead of moved.
298:     let x = Bar(2);
299:     let y = x;
300:     let z = x;
301:
302:     // All integer and float types are ‘Copy‘.
303:     let x = 1;
304:     let y = x;
305:     let z = x;
306:
307:     // So are references.
308:     let a = &x;
309:     let b = a;
310:     let c = a;
311:
312:     ////////////////////////////////////
313:     // 7. "Object-Oriented" Programming //
314:     ////////////////////////////////////
315:
316:     #[derive(Debug)]
317:     struct Point {
318:         x: i32,
319:         y: i32
320:     }
321:
322:     // C-style OOP
323:     fn point_add(a: Point, b: Point) -> Point {
324:         Point { x: a.x + b.x, y: a.y + b.y }
325:     }
```

```
326:
327:     // Java-style OOP
328:     impl Point {
329:         pub fn new(x: i32, y: i32) -> Point {
330:             Point { x, y }
331:         }
332:
333:         // '&self': an immutable reference to 'self'
334:         // Can read 'self' but not write to it.
335:         pub fn add(&self, other: Point) -> Point {
336:             Point { x: self.x + other.x, y: self.y + other.y }
337:         }
338:
339:         // '&mut self': a mutable reference to 'self'
340:         // Can read and write to 'self'.
341:         pub fn set_x(&mut self, x: i32) {
342:             self.x = x;
343:         }
344:     }
345:
346:     // 'mut' is needed to create an '&mut' reference
347:     let mut p1 = Point::new(5, 2);
348:
349:     // the '&mut' reference is automatically created on method call
350:     p1.set_x(10);
351:
352:     let p2 = Point::new(3, 1);
353:     println!("{:?}", p1.add(p2));
354: }
```