Errata for *Rust for Rustaceans* (updated to 3rd printing)

**Page 17:** The end of the paragraph that reads:
“. . . and it reports that the list is still mutably borrowed.”
should now read:
“. . . and it reports that `s` is still mutably borrowed.”

**Page 26:** In the “Non-Generic Inner Functions” info box in the “Traits and Trait Bounds” section, we deleted the word “instead” where duplicated so it now reads: “. . . you can instead declare such a helper function outside the method.”

**Page 45:** The paragraph beginning “Of course, in the real world” now reads:
“Sometimes, you don’t know if your code must own data or not, as it is runtime dependent. For this, the `Cow` type is your friend. It lets you represent data that *may* be owned by holding either a reference or an owned value. If asked to produce an owned value when it only has a reference, a `Cow` uses the `ToOwned` trait to make one behind the scenes, usually by cloning. `Cow` is typically used in return types to represent functions that sometimes allocate. For example, `String::from_utf8_lossy` allocates only if the input contains invalid UTF-8. `Cow` can also be used in arguments for functions that can sometimes make use of owned inputs, but that’s rarer in practice.”

**Page 158:** In Listing 9-5, the statement starting `let init:` should be indented by one more level, so the full listing reads:
```rust
fn fill(gen: impl FnMut() -> Option<u8>) {
    let mut buf = [MaybeUninit::uninit(); 4096];
    let mut last = 0;
    for (i, g) in std::iter::from_fn(gen).take(4096).enumerate() {
        buf[i] = MaybeUninit::new(g);
        last = i + 1;
    }
    // Safety: all the u8s up to last are initialized.
    let init: &[u8] = unsafe {
        MaybeUninit::slice_assume_init_ref(&buf[..last])
    };
    // . . . do something with init . . .
}
```
And in Listing 9-6, the line:

```rust
self.set_len(start + n);
```

should now read:

```rust
self.set_len(start + fill);
```

**Page 162:** In Listing 9-11, the statement that reads:

```rust
unsafe impl<#[may_dangle] T> for Box<T> { /* ... */ }
```

should now read:

```rust
unsafe impl<#[may_dangle] T> Drop for Box<T> { /* ... */ }
```

**Page 202:** In the box “The Niche Optimization in FFI,” the sentence that reads:

“For example, a nullable function pointer can be represented as `Option<extern fn(...)`, and a nullable data pointer can be represented as `Option<*mut T>`.”

should now read:

“For example, a nullable function pointer can be represented as `Option<extern fn(...)`, and a nullable data pointer can be represented as `Option<NotNull<T>>`.”

**Page 235:** In Listing 13-3, the lines:

```rust
impl Drop for DropGuard<'_> { 
    fn drop(&mut self) { 
        lock.store(true, Ordering::Release);
    }
}
```

should now read:

```rust
impl Drop for DropGuard<'_> { 
    fn drop(&mut self) { 
        self.0.store(true, Ordering::Release);
    }
}
```