

# Chapter 8. Iteration and looping

---

## Learn about

- Iterating the items of an array
  - Using the for statement to loop through an array
- 

## Looping through the elements

As a result of the previous chapter about tuples and arrays, we have a number of events in an array, represented by pairs (tuples with two items in each). We can make more events in the same form and put them in the array. The Rust compiler will allocate memory for them on the stack when the program is run.

A Rust array knows its own length, and as we saw briefly in the previous chapter, it can be queried with the `len` method. Let's add a few more events, representing the latest that has happened in Rust, Python, and Java at the time of this writing, and then print them all using a `while` loop:

---

### Example 8.1. Iterating the elements of an array with a while loop

```
fn main() {
    let events = [
        (1996_01_23, "JDK 1.0 released"),
        (2008_12_03, "Python 3.0 released"),
        (2015_05_15, "Rust 1.0.0 released"),
        (2025_09_16, "Java 25 released"),
        (2025_10_07, "Python 3.14 released"),
        (2025_12_11, "Rust 1.92.0 released"),
    ];
    let mut index = 0;
    while index < events.len() {
        println!("[{}] {}: {}", index + 1, events[index].0, events[index].1);
        index += 1;
    }
}
```

```
}
```

---

The output is predictable; all the events printed in exactly the order they occur in the array:

```
[1] 19960123: JDK 1.0 released  
[2] 20081203: Python 3.0 released  
[3] 20150515: Rust 1.0.0 released  
[4] 20250916: Java 25 released  
[5] 20251007: Python 3.14 released  
[6] 20251211: Rust 1.92.0 released
```

The while loop uses the variable `index` to indicate an array element, starting from zero up to one less than the array length. This approach works, but it has some inconveniences. You need:

- to know that the indexing starts at zero
- to have a dedicated index variable
- to manually increment the index on every loop round, and
- make the while loop run only until the indexes run out, to avoid an out-of-bounds error.

This kind of loop is not typically used in Rust when processing *every* item in a collection such as an array.

The traditional `for` loop, as found in "curly-bracket" languages (like C, C++, C#, Java, or JavaScript) is not the typical Rust approach either. That would also mean maintaining an index. Instead, Rust uses *iterators* with the `for` keyword. They work much like the enhanced `for` loop in Java.

An *iterator* is a construct that is implemented by the collection data types in Rust. We can also get an iterator for our own types by implementing the `Iterator` trait. To actually use an iterator in a `for` loop, we need to pick a name for the variable that will hold each item in the collection in turn, and then call the `iter` method of the collection:

---

**Example 8.2. Using an iterator and the for statement on the array**

```
fn main() {
    let events = [
        (1996_01_23, "JDK 1.0 released"),
        (2008_12_03, "Python 3.0 released"),
        (2015_05_15, "Rust 1.0.0 released"),
        (2025_09_16, "Java 25 released"),
        (2025_10_07, "Python 3.14 released"),
        (2025_12_11, "Rust 1.92.0 released"),
    ];

    // With index number:
    /*
    for (index, event) in events.iter().enumerate() {
        println!("[{}] {}: {}", index + 1, event.0, event.1);
    }
    */

    for event in events {
        println!("{}", event.0, event.1);
    }
}
```

---

The program prints out every event in the array:

```
19960123: JDK 1.0 released
20081203: Python 3.0 released
20150515: Rust 1.0.0 released
20250916: Java 25 released
20251007: Python 3.14 released
20251211: Rust 1.92.0 released
```

Actually, we cheated a little, since with the iterator we lost the counter that was in the while version. Not to worry—we can get the index of each element by using the `enumerate` method of the iterator. The indexed version of the for loop then looks like this:

```
for (index, event) in events.iter().enumerate() {
    println!("[{}] {}: {}", index + 1, event.0, event.1);
}
```

The `enumerate` method returns pairs (tuples of two items) with the index of the current item and the actual element. As we will see later on, method chaining in the style of `events.iter().enumerate()` is a common idiom in Rust.

If we only need the element, and not its index number, we don't actually have to explicitly get the iterator. The version without the index number could be written like this:

```
for event in events {  
    println!("{}", event.0, event.1);  
}
```

---

## Chapter summary

Items in an array can be processed with either a `while` loop, maintaining an index, or with a `for` loop, using an iterator we get with the `iter` method of the array. If we need array indices with a `for` loop, we use the `enumerate` method of the iterator.