Input classifier

In PHP

You would walk a list, skip the holes with `??` or a null check, and tally with an associative array. This exercise is the Rust version of that tally. No enums and no structs. Those are chapter 4.

classify.php

```php
 <?php

$inputs = [12, null, 0, -3, 101, null];
$counts = [];

foreach ($inputs as $input) {
    if ($input === null) {
        $counts['missing'] = ($counts['missing'] ?? 0) + 1;
        continue;
    }

    $label = match (true) {
        $input >= 100 => 'large',
        $input >= 1 && $input < 100 => 'positive',
        $input === 0 => 'zero',
        default => 'negative',
    };
    $parity = $input % 2 === 0 ? 'even' : 'odd';
    $counts[$label] = ($counts[$label] ?? 0) + 1;
    $counts[$parity] = ($counts[$parity] ?? 0) + 1;
}

ksort($counts);
foreach ($counts as $label => $count) {
    echo "$label $count\n";
}
```

In Rust

The list is a `Vec<Option<i32>>`. Three functions are yours to fill in. `classify` uses a guard and ranges:

- `n if n >= 100` is `large`

- `1..100` is `positive`
- `0` is `zero`
- anything else is `negative`

`parity` is `even` or `odd`. `summarize` counts each label in a `BTreeMap`, including `missing` for `None`. `&[Option<i32>]` reads the list without taking it, and `&inputs` lends the `Vec`. `Some(&1)` in the tests is a lent count. Chapter 3 explains `&`. `101` is odd, so this list has two evens (`12` and `0`) and two odds (`-3` and `101`).

`cargo test` fails until the three functions do that. `cargo run` then prints the counts in sorted order. The button loads the starter. The solution is collapsed under it.

src/main.rs

```
 use std::collections::BTreeMap;

fn classify(n: i32) -> &'static str {
    todo!("classify {n}")
}

fn parity(n: i32) -> &'static str {
    todo!("parity {n}")
}

fn summarize(inputs: &[Option<i32>]) -> BTreeMap<&'static str, i32> {
    todo!("summarize {}", inputs.len())
}

fn main() {
    let inputs: Vec<Option<i32>> = vec![Some(12), None, Some(0), Some(-3),
Some(101), None];
    let counts = summarize(&inputs);
    for (label, count) in &counts {
        println!("{label} {count}");
    }
}

#[cfg(test)]
mod tests {
    use super::{classify, parity, summarize};

    #[test]
    fn classifies_sign_and_size() {
        assert_eq!(classify(12), "positive");
        assert_eq!(classify(0), "zero");
        assert_eq!(classify(-3), "negative");
        assert_eq!(classify(99), "positive");
        assert_eq!(classify(100), "large");
        assert_eq!(classify(101), "large");
```

```
    }

    #[test]
    fn classifies_parity() {
        assert_eq!(parity(12), "even");
        assert_eq!(parity(0), "even");
        assert_eq!(parity(-3), "odd");
    }

    #[test]
    fn counts_categories() {
        let inputs = vec![Some(12), None, Some(0), Some(-3), Some(101), None];
        let counts = summarize(&inputs);
        assert_eq!(counts.get("positive"), Some(&1));
        assert_eq!(counts.get("zero"), Some(&1));
        assert_eq!(counts.get("negative"), Some(&1));
        assert_eq!(counts.get("large"), Some(&1));
        assert_eq!(counts.get("missing"), Some(&2));
        assert_eq!(counts.get("even"), Some(&2));
        assert_eq!(counts.get("odd"), Some(&2));
    }
}
```

Solution

src/main.rs

```
 use std::collections::BTreeMap;

fn classify(n: i32) -> &'static str {
    match n {
        n if n >= 100 => "large",
        1..100 => "positive",
        0 => "zero",
        _ => "negative",
    }
}

fn parity(n: i32) -> &'static str {
    if n % 2 == 0 { "even" } else { "odd" }
}

fn summarize(inputs: &[Option<i32>]) -> BTreeMap<&'static str, i32> {
    let mut counts = BTreeMap::new();
    for input in inputs {
        match input {
            None => {
                *counts.entry("missing").or_insert(0) += 1;
            }
            Some(n) => {
                *counts.entry(classify(*n)).or_insert(0) += 1;
                *counts.entry(parity(*n)).or_insert(0) += 1;
            }
        }
```

```
    }
    counts
}

fn main() {
    let inputs: Vec<Option<i32>> = vec![Some(12), None, Some(0), Some(-3),
Some(101), None];
    let counts = summarize(&inputs);
    for (label, count) in &counts {
        println!("{label} {count}");
    }
}

#[cfg(test)]
mod tests {
    use super::{classify, parity, summarize};

    #[test]
    fn classifies_sign_and_size() {
        assert_eq!(classify(12), "positive");
        assert_eq!(classify(0), "zero");
        assert_eq!(classify(-3), "negative");
        assert_eq!(classify(99), "positive");
        assert_eq!(classify(100), "large");
        assert_eq!(classify(101), "large");
    }

    #[test]
    fn classifies_parity() {
        assert_eq!(parity(12), "even");
        assert_eq!(parity(0), "even");
        assert_eq!(parity(-3), "odd");
    }

    #[test]
    fn counts_categories() {
        let inputs = vec![Some(12), None, Some(0), Some(-3), Some(101), None];
        let counts = summarize(&inputs);
        assert_eq!(counts.get("positive"), Some(&1));
        assert_eq!(counts.get("zero"), Some(&1));
        assert_eq!(counts.get("negative"), Some(&1));
        assert_eq!(counts.get("large"), Some(&1));
        assert_eq!(counts.get("missing"), Some(&2));
        assert_eq!(counts.get("even"), Some(&2));
        assert_eq!(counts.get("odd"), Some(&2));
    }
}
```

The starter panics on `todo!` until you replace the bodies. The solution's `cargo run` prints the same seven lines the PHP tally prints after `ksort`, in sorted order:

```
 even 2
large 1
missing 2
negative 1
odd 2
positive 1
zero 1
```

> **Watch for:** `101` is odd. `0` is even. `1..100` stops at 99, so 100 falls through to the guard. Write `1..=100` and 100 becomes `positive`.

> **Idiomatic Rust:** `BTreeMap` so the summary is sorted. `match` on `Option`. A guard for the large band, and a range for the rest of the positives.

Side by side

	PHP	Rust
A hole in the list	`null`	`None`
A tally	an associative array	`BTreeMap`
Sorted keys	`ksort`	the map is already sorted

Try it

Open the starter `src/main.rs` in the Playground, or paste it locally and run `cargo test`. The three tests fail on `todo!`. Replace `classify`, `parity`, and `summarize`. `cargo test` passes, and `cargo run` prints the seven sorted lines above.

Key takeaways

- The list is `Vec<Option<i32>>`. The tally is a `BTreeMap`.
- `classify` uses a guard and `1..100`. `parity` is even or odd. `summarize` counts `missing` for `None`.
- `cargo test` is the check. The printout is sorted because the map is a `BTreeMap`.

What's next: What you learned